A Novel Cost-Based Model for Data Repairing

Shuang Hao Nan Tang Guoliang Li Jian He Na Ta Jianhua Feng

Abstract—Integrity constraint based data repairing is an iterative process consisting of two parts: detect and group errors that violate given integrity constraints (ICs); and modify values inside each group such that the modified database satisfies those ICs. However, most existing automatic solutions treat the process of detecting and grouping errors straightforwardly (*e.g.,* violations of functional dependencies using string equality), while putting more attention on heuristics of modifying values within each group. In this paper, we propose a revised semantics of *violations* and *data consistency w.r.t.* a set of ICs. The revised semantics relies on string similarities, in contrast to traditional methods that use syntactic error detection using string equality. Along with the revised semantics, we also propose a new cost model to quantify the cost of data repair by considering distances between strings. We show that the revised semantics provides a significant change for better detecting and grouping errors, which in turn improves both precision and recall of the following data repairing step. We prove that finding minimum-cost repairs in the new model is NP-hard, even for a single FD. We devise efficient algorithms to find approximate repairs. In addition, we develop indices and optimization techniques to improve the efficiency. Experiments show that our approach significantly outperforms existing automatic repair algorithms in both precision and recall.

Index Terms—Data Repairing, Functional Dependencies, Fault-Tolerant Violation, Graph Model, Maximal Independent Set

1 Introduction

Data cleaning, which is to detect and repair data errors, has played an important part in the history of data management, because high-quality business decisions must be made based on high-quality data, especially in the era of big data [26], [30], [31].

Although involving users [18], [25], [29], [37] in data repairing is important in practice, automatic repairing [3], [6], [11], [13], [34], [38] is still valuable. Because when users do not have enough capacity to identify and repair data errors, automatic approaches can help find possible repairs, which can alleviate the burden of users in data cleaning.

Automatic data repairing typically consists of two parts: detect and group errors that violate given integrity constraints (ICs); and modify values inside each group such that the modified database satisfies those ICs.

Example 1: Consider a table $\mathcal{D}$ for US citizens of schema Citizens (Name, Education, Level, City, Street, District, State). A Citizens instance is shown in Table 1. Tuples t_1–t_5 (resp. t_6–t_{10}) are about citizens in New York (resp. Boston). Both errors and their correct values are highlighted. For instance, t_4[State] = MA is wrong, whose correct value is NY. □

These data errors are normally detected by integrity constraints (ICs) such as functional dependencies (FDs).

Example 2: Consider $\mathcal{D}$ in Example 1 and three FDs:

- *Shuang Hao is with the Department of Computing Science, Tsinghua University, Beijing, China.haos13@mails.tsinghua.edu.cn.*
- *Nan Tang is with Qatar Computing Research Institute, Hamad Bin Khalifa Univeristy, Qatar. ntang@hbku.org.qa.*
- *Guoliang Li is with the Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China. liguoliang@tsinghua.edu.cn.*
- *Jian He is with the Department of Computing Science, Tsinghua University, Beijing, China. hej13@mails.tsinghua.edu.cn.*
- *Na Ta is with the Department of Computing Science, Tsinghua University, Beijing, China. dan13@mails.tsinghua.edu.cn. Corresponding Author.*
- *Jianhua Feng is with the Department of Computing Science, Tsinghua University, Beijing, China. fengjh@tsinghua.edu.cn.*

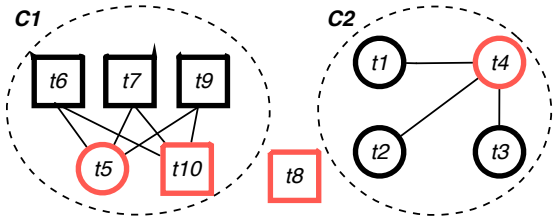


Fig. 1. Violations of φ_2 : [City] → [State]

φ_1: Citizens ([Education] → [Level])
φ_2: Citizens ([City] → [State])
φ_3: Citizens ([City, Street] → [District])

φ_1 denotes that Education uniquely determines educational Level. The two tuples t_1 and t_9 violate φ_1, as they have the same Education (*i.e.,* Bachelors) but different Level values (13 for t_1 and 10 for t_9). It is similar for φ_2 and φ_3. Data repairing using these FDs typically perform the following two consecutive steps, taking φ_2 for example.

Error detection. Violations are detected if two tuples having the same City but different State, which are depicted in Fig. 1. Each node represents a tuple (*e.g.,* t_6) and each edge represents a violation between two tuples (*e.g.,* (t_5, t_6) is a violation *w.r.t.* φ_2). Black nodes (resp. red nodes) represent correct (resp. erroneous) tuples. Circular (resp. rectangular) nodes are citizens in New York (resp. Boston).

Data repairing. Violation graph is naturally partitioned in the concept of connected components. Naturally, a data repairing algorithm will treat each connected component independently and modify values locally inside each connected component to resolve the inconsistency. Consider C_1 in Fig. 1 for example. When considering $t_5, t_6, t_7, t_9, t_{10}$ together, most algorithms will change t_5[State] and t_{10}[State] from NY to MA to compute a consistent database. The latter change is correct (for t_{10}[State]), but the former change is wrong (for t_5[State]). □

Example 2 reveals two shortcomings of existing automatic data repairing algorithms [5], [6], [11], [13], [19].

TABLE 1
Example instance of US citizens (φ_1 : [Education] $\rightarrow$ [Level], φ_2 : [City] $\rightarrow$ [State], φ_3 : [City, Street] $\rightarrow$ [District])

	Name	Education	Level	City	Street	District	State
t_1 :	Janaina	Bachelors	13	New York	Main	Manhattan	NY
t_2 :	Aloke	Bachelors	13	New York	Main	Manhattan	NY
t_3 :	Jieyu	Bachelors	13	New York	Western	Queens	NY
t_4 :	Paulo	Masters	14	New York	Western	Queens	MA $\rightarrow$ NY
t_5 :	Zoe	Masters	14	Boston $\rightarrow$ New York	Main	Manhattan	NY
t_6 :	Gara	Masers $\rightarrow$ Masters	14	Boston	Main	Financial	MA
t_7 :	Mitchell	HS-grad	9	Boston	Main	Financial	MA
t_8 :	Pavol	Masters	13 $\rightarrow$ 14	Boton $\rightarrow$ Boston	Arlingto	Brookside	MA
t_9 :	Thilo	Bachelors	10 $\rightarrow$ 13	Boston	Arlingto	Brookside	MA
t_{10} :	Nenad	Bachelers $\rightarrow$ Bachelors	13	Boston	Arlingto	Brookside	NY $\rightarrow$ MA

(i) *Unrobust error detection:* Some erroneous tuple value (*e.g.*, t_8[City]) cannot be captured. (ii) *Bad error group:* When associating errors with tuples straightforwardly (*e.g.*, t_5 in C_1) to reason about, most algorithms will fail to repair t_5.

The above observations call for a revision of data repairing problems that improves both (i) and (ii) discussed above. Intuitively, improving (ii) can increase the precision, and enhancing (i) will advance recall. Let us consider the following example that illutrates the benefits of our new cost-based model.

Example 3: Given the instance $\mathcal{D}$ and FD φ_2 in Table 1, as t_8 is not in conflict with any other tuple, the error in t_8[City] can not be detected in the above example. Thus, our approach relaxes the conditions of capturing violations, *e.g.*, from syntactic equality string matching to more semantic similarity string matching. Then, since t_8[City$\cup$State] is very similar to t_9[City $\cup$ State], we consider them as a violation *w.r.t.* φ_2 and t_8[City] will be modified to Boston.

Besides, we jointly consider multiple constraints to minimize the repair cost. When repairing tuple t_5, φ_3 is also taken into consideration. Through maximal independent sets, we make sure that (New York, NY), (Boston, MA) are correct *w.r.t.* φ_2 and (New York, Main, Manhattan), (Boston, Main, Financial) are correct *w.r.t.* φ_3. We join them and get (New York, Main, Manhattan, NY), (Boston, Main, Financial, MA). Obviously, repairing t_5[City] has the minimal repairing cost and meanwhile eliminates both violations of φ_2 and φ_3. □

Contributions. We make the following contributions.

(1) We propose a novel cost-based model for data repairing. We show that finding the optimal solution for this problem is NP-hard (Section 2).

(2) We devise new data repairing algorithms. Specifically, we present algorithms to handle a single constraint (Section 3) as well as multiple constraints (Section 4).

(3) We study optimization techniques to improve the efficiency of data repairing (Section 5).

(4) We conduct extensive experiments (Section 6), which demonstrate that our method outperforms *state-of-the-art* automatic data repairing approaches in accuracy.

2 Problem Formulation

Consider an instance $\mathcal{D}$ of relation $\mathcal{R}$. To simplify the discussion, we focus on functional dependencies (FDs), and both theoretical results and algorithms can be applied on its extension, conditional functional dependencies (CFDs) [15].

2.1 Functional Dependencies and Semantics

We use a standard definition for an FD φ: $X \rightarrow Y$. We say that $\mathcal{D}$ *satisfies* φ, denoted by $\mathcal{D} \models \varphi$, if for any two tuples (t_1, t_2) in $\mathcal{D}$, when $t_1[X] = t_2[X]$, $t_1[Y] = t_2[Y]$ holds. Otherwise, we call (t_1, t_2) a violation of φ, if $t_1[X] = t_2$[X] but $t_1[Y] \neq t_2[Y]$.

Moreover, we say that $\mathcal{D}$ is *consistent w.r.t.* a set Σ of FDs, denoted by $\mathcal{D} \models \Sigma$, if $\mathcal{D} \models \varphi$ for each $\varphi \in \Sigma$.

Example 4: Consider the FD φ_1 : [Education] $\rightarrow$ [Level] in Example 1 and the instance $\mathcal{D}$ in Table 1. $(t_4, t_8) \not\models \varphi_1$, since t_4[Education] $= t_8$[Education] = Masters, but t_4[Level] $\neq t_8$[Level]. Thus, we have $\mathcal{D} \not\models \varphi_1$. Note that $(t_4, t_6) \models \varphi_1$ because t_4[Education] $\neq t_6$[Education] □

To address the two limitations of automatic repairing algorithms discussed in Section 1, we use a similarity function to holistically compare both left and right hand side of constraints. More specifically, given an FD φ: $X \rightarrow Y$, we use a distance function $\mathsf{dist}_\varphi()$ to capture violations. We simply write $\mathsf{dist}()$ when φ is clear from the context. We also write t^φ for $t[X \cup Y]$.

Distance function. For attribute A in $\mathcal{R}$, $\mathsf{dist}(t_1[A], t_2[A])$ indicates how similar $t_1[A]$ and $t_2[A]$ are. There are many known distance functions, *e.g.*, Edit distance, Jaccard distance, and Euclidean distance. In this paper, by default, we use edit distance if $t_1[A]$ and $t_2[A]$ are strings and Euclidean distance if the values are numeric. Any other distance function can also be used. Formally, we have:

$$\mathsf{dist}(t_1[A], t_2[A]) = \begin{cases} Edit(t_1[A], t_2[A]) & \text{string} \\ Eucli(t_1[A], t_2[A]) & \text{numeric} \end{cases} \quad (1)$$

where $\mathsf{dist}(t_1[A], t_2[A])$ is a normalized distance in [0,1].

For t_1^φ and t_2^φ with multiple attributes, their distance is:

$$\mathsf{dist}(t_1^\varphi, t_2^\varphi) = w_l \sum_{A_l \in X} \mathsf{dist}(t_1[A_l], t_2[A_l]) + w_r \sum_{A_r \in Y} \mathsf{dist}(t_1[A_r], t_2[A_r]). \quad (2)$$

where w_l, w_r are weight coefficients in [0,1] and $w_l + w_r = 1$.

Example 5: Consider table $\mathcal{D}$ and FD φ_1 in Table 1. The distance between t_4 and t_6 is $\mathsf{dist}(t_4^{\varphi_1}, t_6^{\varphi_1}) = 0.5 \times \mathsf{dist}(t_4[\text{Education}], t_6[\text{Education}]) + 0.5 \times \mathsf{dist}(t_4[\text{Level}], t_6[\text{Level}]) = 0.5 \times 0.14 + 0 = 0.07$. □

Fault-tolerant violation. Two tuples (t_1, t_2) are in a fault-tolerant (FT-) violation *w.r.t.* φ, if (1) $t_1^\varphi \neq t_2^\varphi$; and (2) the distance between them is no larger than a given threshold τ, *i.e.*, $\mathsf{dist}(t_1^\varphi, t_2^\varphi) \leq \tau$. We write $(t_1, t_2) \not\models^* \varphi$ if both (1) and (2) hold (*i.e.*, an FT-violation); otherwise, we write $(t_1, t_2) \models^* \varphi$

if $\mathsf{dist}(t_1^{\varphi}, t_2^{\varphi}) > \tau$. A possible method of deciding the threshold τ is as follows. We first calculate the distance of each pair of tuples $(t_1^{\varphi}, t_2^{\varphi})$ and sort them in ascending order. When the difference between the two adjacent numbers suddenly becomes large, we choose the smaller value τ as the threshold. Besides, if precision rather than recall is regarded as the more important criterion, we can conservatively decrease threshold τ further.

Remark. The first benefit of using FT-violation is: it captures errors that cannot be detected by standard FD semantics using string equalities, *e.g.,* $t_8[\mathsf{City}]$. Better still, it can associate errors with correct tuples. Taking the error $t_5[\mathsf{City}]$ and φ_2 for instance, traditional methods will associate t_5 with t_6, t_7, t_9, t_{10} since they have the same City Boston. In contrast, we will associate t_5 with t_1–t_4, which will naturally result in a higher quality repair.

Note that, if we set $w_l = 1, w_r = 0$ and threshold $\tau = 0$, fault-tolerant (FT-) violation degrades to the traditional FD violation. If we increase threshold τ, we can detect typos in the left hand. In this case, we cannot ignore the distance in the right hand of FD. Two tuples that have similar left hand but very different right hand are not violation. Thus, we can control the percentage of right hand distance through weight w_r. In this paper, by default, we set $w_l = w_r = 0.5$.

Fault-tolerant consistency. A database $\mathcal{D}$ is fault-tolerant (FT-) *consistent* to an FD φ, denoted by $\mathcal{D} \models^{*} \varphi$, if there do not exist two tuples t_1, t_2 such that $(t_1, t_2) \not\models^{*} \varphi$. We say that $\mathcal{D}$ is FT-consistent *w.r.t.* a set Σ of FDs, denoted by $\mathcal{D} \models^{*} \Sigma$, if $\mathcal{D} \models^{*} \varphi$ for each $\varphi \in \Sigma$.

Example 6: Consider instance $\mathcal{D}$ and FD φ_1 in Table 1. Let the threshold $\tau = 0.35$. Since $\mathsf{dist}(t_4^{\varphi_1}, t_6^{\varphi_1}) = 0.07 < \tau$, we have $(t_4, t_6) \not\models^{*} \varphi_1$ and $\mathcal{D} \not\models^{*} \varphi_1$. The error in $t_6[\mathsf{Education}]$ can be captured and $t_6[\mathsf{Education}]$ is repaired to Masters. □

Consistency vs FT-consistency. The FT-consistency semantics means that if two tuples on attributes $X \cup Y$ are similar but not identical, they are FT-violated and should be repaired. It is readily to see that in the case of $\tau \geq w_r|Y|$ (where $|Y|$ is the number of attributes in Y), if a database $\mathcal{D}$ is FT-consistent *w.r.t.* φ, it must be also consistent. Considering $w_l = 1, w_r = 0, \tau = 0$, if $\mathcal{D}$ is consistent, it must be also FT-consistent and vice versa.

Theorem 1: *Given a database $\mathcal{D}$ and an FD $\varphi : X \rightarrow Y$, when $\tau \geq w_r|Y|$, if $\mathcal{D} \models^{*} \varphi$, then $\mathcal{D} \models \varphi$.* □

Proof sketch: If (t_1, t_2) is a violation *w.r.t.* φ, which means $t_1[X] = t_2[X]$ but $t_1[Y] \neq t_2[Y]$, their distance $\mathsf{dist}(t_1^{\varphi}, t_2^{\varphi})$ cannot be larger than $w_r|Y|$. Thus when $\tau \geq w_r|Y|$, (t_1, t_2) is also a FT-violation *w.r.t.* φ. In other words, if $\mathcal{D}$ is FT-consistent when $\tau \geq w_r|Y|$, any two tuples t_1, t_2 satisfy $\mathsf{dist}(t_1^{\varphi}, t_2^{\varphi}) > \tau \geq w_r|Y|$. So $\mathcal{D}$ must be consistent. □

2.2 Problem Statement

Close-world data repair model. The repaired value for an attribute A must come from the active domain of A.

Valid tuple repair. Repairing a tuple from t to t' (t' may not be in $\mathcal{D}$ originally) is called a *valid tuple repair*, if for any FD φ, there exists a tuple t'' in $\mathcal{D}$ such that $t'^{\varphi} = t''^{\varphi}$. In other words, the whole tuple t' may be new to $\mathcal{D}$; however, the projected values t'^{φ} must exist in $\mathcal{D}$ originally.

For example, consider tuple t_6 and φ_1 in Table 1. Currently, $t_6^{\varphi_1} = \{\text{Masers}, 14\}$. A repair to $\{\text{Masters}, 14\}$ is valid since the values exist in t_4. However, a repair to $\{\text{Bachelors}, 14\}$ is not valid, since they are new to the table.

Valid database repair. A database $\mathcal{D}'$ is a *valid database repair* of $\mathcal{D}$ *w.r.t.* a set Σ of FDs in FT-consistent semantics, if $\mathcal{D}'$ is FT-consistent *w.r.t.* Σ only via valid tuple repairs.

Repair cost. For each tuple t in $\mathcal{D}$, suppose t' is the corresponding tuple in the repaired database $\mathcal{D}'$. Obviously, if $t = t'$, the repair cost is 0; otherwise we use the distance functions to quantify the repair cost as below.

$$\mathsf{cost}(t, t') = \sum_{A \in \mathcal{R}} \mathsf{dist}(t[A], t'[A]). \tag{3}$$

Naturally, the repair cost of database $\mathcal{D}$ is

$$\mathsf{cost}(\mathcal{D}, \mathcal{D}') = \sum_{t \in \mathcal{D}} \mathsf{cost}(t, t'). \tag{4}$$

For example, assume that tuple t_{10} is repaired as illustrated in Table 1, the repair cost $\mathsf{cost}(t_{10}, t'_{10}) = \mathsf{dist}(t_{10}[\mathsf{Education}], t'_{10}[\mathsf{Education}]) + \mathsf{dist}(t_{10}[\mathsf{State}], t'_{10}[\mathsf{State}]) = 0.11 + 1 = 1.11$.

Problem statement. Given a database $\mathcal{D}$ and a set Σ of FDs defined on $\mathcal{D}$, the *data repairing problem* is to find a valid repair $\mathcal{D}'$ of $\mathcal{D}$ such that $\mathcal{D}'$ is FT-consistent and $\mathsf{cost}(\mathcal{D}, \mathcal{D}')$ is minimum among all valid repaired databases.

2.3 Related Work

Many works use integrity constraints in cleaning data (*e.g.,* [1], [7], [9], [15], [16], [23]; see [14] for a survey). They have been revisited to better capture data errors as violations of these constraints (*e.g.,* CFDs [15] and CINDs [7]). We categorize related work as follows.

Constraint-based data repairing. Automatic data repairing algorithms based on integrity constraints have been proposed [5], [6], [11], [13], [19]. Heuristic methods are developed in [4], [6], [19], based on FDs [4], [23], FDs and INDs [6], CFDs [15], CFDs and MDs [17], and denial constraints [11]. Several algorithms take advantage of confidence values, which are placed by users, to guide the repairing process [12], [17]. The most related studies are metric dependencies [24] and differential dependencies [32], which use similarity functions in either left or right hand of a constraint to relax the matching condition. In contrast to these prior art, we take a more holistic way by considering the similarity matching combining *both* left *and* right hand of attributes of a constraint, which is discussed in Section 2. We will demonstrate by experiments (Section 6) that the new way of reasoning about errors can significantly improve both precision and recall of data repairing, as observed earlier in Example 2.

Rule-based data repairing. Rule-based data repairing [18], [28], [35] differs from the above constraint-based data repairing in the way to modify a value is explicitly encoded in the rule, hence is deterministic. However, this requires external information about the golden standard, which normally comes from domain experts, or master data.

Of course, rule-based data repairing is preferred. However, when such external information is not available, or not enough, constraint-based approaches are needed to (approximately) discover *golden standard* from the data at hand.

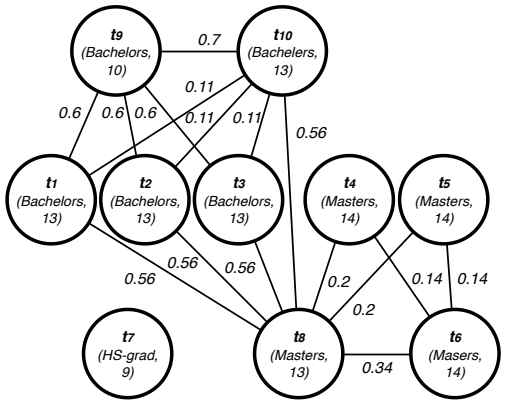


Fig. 2. Graph model of φ_1

User guided data repairing. Several studies [10], [18], [21], [27], [37] have involved (crowd) users to improve the precision of data repairing. However, involving users is typically painful and error-prone, and the required domain experts are often unavailable (for crowd users) or without enough capacity.

As remarked earlier, we target at automatic data repairing in this work, which is complementary to user guided repairing when users are unavailable to fulfill the work.

Machine learning and statistical cleaning. There are machine learning based data repairing methods [36], [37], as well as statistical methods [2], [22]. For ML-based data repairing, they are typically supervised, which heavily depend on both the training datasets and the selected features. Hence, it is hard to have a fair comparison with automatic approaches. For statistical repairing, they normally assume a desired data distribution, and update data values to get closer to the desired distribution. In fact, statistical (or quantitive) data repairing should be combined with integrity constraint-based data repairing, since they have various targets.

3 Single Constraint Solutions

We first study the *single FD repairing problem* where only one FD is present. We prove that it is NP-hard, and propose an expansion-based algorithm to find the optimal result and a greedy algorithm to find an approximate result.

Graph model. We introduce a graph model $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each vertex $v \in \mathcal{V}$ is a tuple, and an undirected edge between two vertices u and v indicates that (u, v) is an FT-violation *w.r.t.* φ. Each edge is associated with a weight, denote by $\omega(u, v) = \mathsf{cost}(u^{\varphi}, v^{\varphi})$. We use vertex and tuple interchangeably if the context is clear. Given a database $\mathcal{D}$, we first transform it to a graph $\mathcal{G}$ and then utilize the graph $\mathcal{G}$ to compute a repair database $\mathcal{D}'$.

Independent set, maximal independent set, and maximum independent set. A subset $\mathcal{I} \subseteq \mathcal{V}$ is an *independent set*, if for any two vertices u, v in $\mathcal{I}$, there is no edge connecting them, *i.e.*, $(u, v) \notin \mathcal{E}$. An independent set $\mathcal{I}$ is called a *maximal independent set* if (1) $\mathcal{I} = \mathcal{V}$ or (2) for any vertex $v \in \mathcal{V} \backslash \mathcal{I}$ ("$\backslash$" is for set minus), $\mathcal{I} \cup \{v\}$ is not an independent set. A maximal independent set with the largest number of vertices is called the maximum independent set.

Example 7: Fig. 2 shows a graph $\mathcal{G}$ of FD φ_1. Each vertex is a tuple in Table 1, and an edge represents an FT-violation *e.g.*, (t_1, t_9). The weight of edge (t_1, t_9) is $\omega(t_1, t_9) = 0 + \mathsf{dist}(t_1[\mathsf{Level}], t_9[\mathsf{Level}]) = \frac{13-10}{14-9} = 0.6$. (We normalize the Euclidean distance by dividing the largest distance.) The set $\mathcal{I}_1 = \{t_1, t_2, t_3, t_7\}$ is an independent set, as there is no edge connecting any two of them in $\mathcal{G}$. $\mathcal{I}_1$ is not maximal because $\mathcal{I}_1 \cup \{t_6\}$ is also an independent set. The set $\mathcal{I}_2 = \{t_1, t_2, t_3, t_6, t_7\}$ is a maximal independent set, as no vertex $v \in \mathcal{G}$ can be added to $\mathcal{I}_2$ and $\mathcal{I}_2 \cup \{v\}$ is still an independent set. The set $\mathcal{I}_3 = \{t_1, t_2, t_3, t_4, t_5, t_7\}$ is the maximum independent set because there is no maximal independent set with more vertices. □

Repairing based on a maximal independent set. A maximal independent set $\mathcal{I}^*$ has some salient properties. (1) Tuples in $\mathcal{I}^*$ have no FT-violations and are thus FT-consistent. (2) Any tuple that is not in $\mathcal{I}^*$ must have FT-violations with at least one tuple in $\mathcal{I}^*$. Based on these two features, we can repair the database using a maximal independent set as follows. Given a maximal independent set $\mathcal{I}^*$, for any tuple x not in $\mathcal{I}^*$, let $\mathcal{N}(x) = \{v | v \in \mathcal{I}^* \text{ and } (v, x) \in \mathcal{E}\}$ denote the neighbor set of x in $\mathcal{I}^*$. We can repair x to any tuple $v \in \mathcal{N}(x)$ (by modifying x^{φ} to v^{φ} in order to resolve the violation between them) and the repairing cost is $\omega(x, v)$. Naturally we want to repair x to v with the minimal cost, *i.e.*, $\omega(x, v) \leq \omega(x, u)$ for any $u \in \mathcal{N}(x)$, and the cost of repairing x given $\mathcal{I}^*$ is $\mathsf{cost}(x|\mathcal{I}^*) = \omega(x, v)$. We enumerate every tuple not in $\mathcal{I}^*$, and iteratively repair all tuples in $\mathcal{D}$ using $\mathcal{I}^*$ and get a repaired database $\mathcal{D}'$. The total cost of repairing $\mathcal{D}$ given $\mathcal{I}^*$ is $\mathsf{cost}(\mathcal{D}, \mathcal{D}') = \mathsf{cost}(\mathcal{D}|\mathcal{I}^*) = \sum_{x \in (\mathcal{V} \backslash \mathcal{I}^*)} \mathsf{cost}(x|\mathcal{I}^*)$.

Optimal Repairing. There may be multiple maximal independent sets. We enumerate every maximal independent set, select the independent set ($\mathcal{I}^B$) with the minimal repairing cost, *i.e.*, $\mathsf{cost}(\mathcal{D}|\mathcal{I}^B) \leq \mathsf{cost}(\mathcal{D}|\mathcal{I}^*)$, which is called the best maximal independent set, and use $\mathcal{I}^B$ to repair the database. We prove that using the best maximal independent set $\mathcal{I}^B$ to repair $\mathcal{D}$ is optimal as stated in Theorem 2.

Theorem 2: *Repairing database $\mathcal{D}$ using the best maximal independent $\mathcal{I}^B$ is optimal.* □

Proof sketch: First, we prove that the repaired database $\mathcal{D}'$ using $\mathcal{I}^B$ is FT-consistent. $\mathcal{I}^B$ is an independent set in the graph model of φ. Thus, for any two vertices u, v in $\mathcal{I}^B$, there is no edge connecting them, which means that the corresponding tuples have no FT-violation *w.r.t.* φ. Furthermore, as $\mathcal{I}^B$ is maximal, any tuple that is not in $\mathcal{I}^B$ must have FT-violations with some tuples in $\mathcal{I}^B$ and be modified to one of them. So $\mathcal{D}'$ is FT-consistent.

Next, we prove that the cost of using $\mathcal{I}^B$ for repairing $\mathcal{D}$ is smallest. Obviously, the optimal repair must correspond to a maximal independent set in the graph model of φ. Given a maximal independent set $\mathcal{I}^*$ and a tuple $x \notin \mathcal{I}^*$, $\mathsf{cost}(x|\mathcal{I}^*) = \{\omega(x, v) | \forall u \in \mathcal{N}(x), \omega(x, v) \leq \omega(x, u)\}$ and $\mathsf{cost}(\mathcal{D}|\mathcal{I}^*) = \sum_{x \notin \mathcal{I}^*} \mathsf{cost}(x|\mathcal{I}^*)$. Thus, $\mathsf{cost}(\mathcal{D}|\mathcal{I}^*)$ is the smallest repair cost of using $\mathcal{I}^*$ to repair $\mathcal{D}$. For each maximal independent set $\mathcal{I}^*$, $\mathsf{cost}(\mathcal{D}|\mathcal{I}^B) \leq \mathsf{cost}(\mathcal{D}|\mathcal{I}^*)$. So the cost of using $\mathcal{I}^B$ for repairing $\mathcal{D}$ is smallest. □

We can prove that the single FD repairing problem is NP-hard as formalized in Theorem 3.

Theorem 3: *Single FD repairing problem is NP-hard.* □

Proof sketch: Consider a special case of the single FD repairing problem: the weight of each edge is equal in the graph model. In this situation, the best maximal independent set is exactly the maximum independent set. This is because the repair cost is proportional to the number of vertices

that do not belong to the independent set. Since finding the maximum independent set is NP-hard, the single FD repairing problem is also NP-hard. □

Repairing Algorithms. Note that existing optimal and approximation algorithms to the maximum independent set problem cannot be used in our problem, as we need to consider the repairing weight. Thus in the following of this section, we will first present an expansion-based algorithm to compute the optimal solution (Section 3.1), and then describe a heuristic but more efficient algorithm (Section 3.2).

3.1 Optimal: Expansion-based Algorithm

To find the optimal solution, we need to compute the best maximal independent set $\mathcal{I}^B$. We propose an expansion-based method with pruning ability to efficiently identify $\mathcal{I}^B$.

Expansion-based algorithm. Obviously, the independent sets satisfy the *a-priori* property, *i.e.*, any subset of an independent set must be an independent set. We access tuples in order, *e.g.*, $t_1, \cdots, t_{|\mathcal{D}|}$. For tuple t_i, we generate all the maximal independent sets of $\mathcal{D}_i = \{t_1, \cdots, t_i\}$. When visiting t_{i+1}, we utilize the maximal independent sets of $\mathcal{D}_i$ to generate those of $\mathcal{D}_{i+1}$. For any maximal independent set $\mathcal{I}$ of $\mathcal{D}_i$, we check if t_{i+1} is FT-consistent with $\mathcal{I}$, *i.e.*, whether t_{i+1} is FT-consistent with every tuple in $\mathcal{I}$. If so, $\mathcal{I} \cup \{t_{i+1}\}$ is a maximal independent set of $\mathcal{D}_{i+1}$; otherwise, we identify the tuples in $\mathcal{I}$ that are FT-consistent with t_{i+1}, denoted by $\texttt{FTC}(t_{i+1}, \mathcal{I})$, and $\texttt{FTC}(t_{i+1}, \mathcal{I}) \cup \{t_{i+1}\}$ is an independent set. Note that if t_{i+1} is not FT-consistent with $\mathcal{I}$, we still need to keep $\mathcal{I}$, as it is also a maximal independent set of $\mathcal{D}_{i+1}$. $\texttt{FTC}(t_{i+1}, \mathcal{I}) \cup \{t_{i+1}\}$ may be not maximal, as it may be a subset of $\texttt{FTC}(t_{i+1}, \mathcal{I}') \cup \{t_{i+1}\}$ where $\mathcal{I}'$ is a maximal independent set of $\mathcal{D}_i$. Thus we need to remove the non-maximal independent sets after expanding every maximal independent set $\mathcal{I}$ of $\mathcal{D}_i$ and then generate the maximal independent sets of $\mathcal{D}_{i+1}$. Iteratively we can generate the maximal independent sets of $\mathcal{D}_{|\mathcal{D}|} = \mathcal{D}$.

Pruning techniques. We discuss the ways to prune some maximal independent sets of $\mathcal{D}_i$ that cannot be expanded to the best independent set. To this end, we estimate a lower bound of using an independent set $\mathcal{I}$ to repair $\mathcal{D}$, denoted by $\texttt{LB}(\mathcal{I})$, and an upper bound $\texttt{UB}(\mathcal{I})$. Given two independent sets $\mathcal{I}$ and $\mathcal{I}'$, if $\texttt{LB}(\mathcal{I}) > \texttt{UB}(\mathcal{I}')$, we prune $\mathcal{I}$ and do not expand it, since using $\mathcal{I}'$ has a smaller repairing cost.

Lower bound. Given an independent set $\mathcal{I}$ of $\mathcal{D}_i$, if we use $\mathcal{I}$ to repair $\mathcal{D}$, the tuples in $\mathcal{D}_i \setminus \mathcal{I}$ must be repaired as they have FT-violations with $\mathcal{I}$ (and other tuples may be FT-consistent with $\mathcal{I}$ and do not need repair), and thus the minimal repairing cost is $\sum_{v \in \mathcal{D}_i \setminus \mathcal{I}} \min_{u \notin \mathcal{D}_i \setminus \mathcal{I}} \omega(u, v)$. If we use another independent set $\mathcal{I}^e$ expanded from $\mathcal{I}$ to repair $\mathcal{D}$, the tuples in $\mathcal{D}_i \setminus \mathcal{I}$ must be repaired, because they have FT-violations with $\mathcal{I}$ (and also $\mathcal{I}^e$), and the minimal repairing cost is still $\sum_{v \in \mathcal{D}_i \setminus \mathcal{I}} \min_{u \notin \mathcal{D}_i \setminus \mathcal{I}} \omega(u, v)$. Thus we can estimate a lower bound of the repairing cost of using $\mathcal{I}$ or independent sets expanded from $\mathcal{I}$ to repair $\mathcal{D}$, *i.e.*,

$$\texttt{LB}(\mathcal{I}) = \sum_{v \in \mathcal{D}_i \setminus \mathcal{I}} \min(\omega(v, u) | u \notin \mathcal{D}_i \setminus \mathcal{I}). \quad (5)$$

Upper bound. The upper bound of using $\mathcal{I}$ to repair $\mathcal{D}$ is to repair every tuple not in $\mathcal{I}$ to a tuple in $\mathcal{I}$ (even though the tuple is FT-consistent with $\mathcal{I}$). It is easy to prove that using $\mathcal{I}^e \supset \mathcal{I}$ to repair $\mathcal{D}$ has smaller cost than using $\mathcal{I}$. Thus we can estimate an upper bound of the repairing cost of using $\mathcal{I}$ or its super independent sets to repair $\mathcal{D}$, *i.e.*,

$$\texttt{UB}(\mathcal{I}) = \sum_{v \in \mathcal{D} \setminus \mathcal{I}} \min(\omega(v, u) | u \in \mathcal{I}). \quad (6)$$

We can leverage the lower bound and upper bound between two independent sets for pruning.

Theorem 4: *Given two independent sets $\mathcal{I}$ and $\mathcal{I}'$, if $\texttt{LB}(\mathcal{I}') > \texttt{UB}(\mathcal{I})$, $\mathcal{I}'$ can be pruned and need not to be expanded.* □

Proof sketch: The repairing cost of using $\mathcal{I}'$ or its descendants in the tree to repair $\mathcal{D}$ cannot be smaller than $\texttt{LB}(\mathcal{I}')$. Meanwhile, $\texttt{UB}(\mathcal{I})$ is the largest repairing cost of using $\mathcal{I}$ or the independent sets contain $\mathcal{I}$ to repair $\mathcal{D}$. $\texttt{LB}(\mathcal{I}') > \texttt{UB}(\mathcal{I})$ means that the independent sets in the subtree rooted at $\mathcal{I}'$ corresponding to larger repairing cost than the maximal independent sets contain $\mathcal{I}$. So $\mathcal{I}'$ can be pruned and does not need not to be expanded. □

Tree-based index. We propose a tree-based algorithm to expand independent sets. The tree has $|\mathcal{D}|$ levels. Each level i corresponds to a tuple t_i. Each node in level i corresponds to a maximal independent set of $\mathcal{D}_i$. The tuple set of the root is $\{t_1\}$. Each node has one or two children.

Tree-based algorithm. Algorithm 1 gives the algorithm using the tree-index. We initialize the tree index with only a root node $\{t_1\}$ (line 1) and the upper bound $\texttt{UB}(\mathcal{T})$ is first assigned to $+\infty$ (line 2). We then visit the tree-index level by level (lines 3-13). For each level, we compute and update the upper bound of the tree (lines 4-5). We also compute the lower bound of each visited node $\mathcal{I}$ (line 7), based on its lower bound and the case of consistency, we decide how to generate the children of the visited node (lines 7-13). If $\mathcal{I} \cup \{t_{i+1}\}$ is also an independent set, node $\mathcal{I}$ has only one child (lines 8-9). Otherwise, we copy this node $\mathcal{I}$ as its left child (line 11). Besides, if $\texttt{FTC}(t_{i+1}, \mathcal{I}) \cup \{t_{i+1}\}$ is a maximal independent set *w.r.t.* $\{t_1, ..., t_{i+1}\}$ and has not appeared in the tree, $\texttt{FTC}(t_{i+1}, \mathcal{I}) \cup \{t_{i+1}\}$ will be the right child of $\mathcal{I}$ (lines 12-13). For the computed maximal independent set (line 14), we repair tuples by using the one with the minimum cost (lines 15-16). The repaired database $\mathcal{D}'$ is returned (line 17).

Accessing order. Note that the order of accessing tuples has no impact on the repair quality, as the best independent set can be enumerated in any order. But the order affects the performance. We can access tuples sorted by their frequencies in the descending order. This is because the maximal independent set with the highest frequent tuples is likely to have small repair cost and thus we generate it earlier and use it to prune other maximal independent sets.

Example 8: Consider the FD φ_1 : [Education] $\rightarrow$ [Level] and $\mathcal{D}$ in Table 1. Some tuples have the same values in both Education and Level (such as t_1, t_2 and t_3), so we just show how the algorithm works on $\{t_1, t_4, t_6, t_7, t_8, t_9, t_{10}\}$. The tree is built as shown in Fig. 3. Some nodes are duplicates and should be removed such as node n_{10}. Node n_{21} is not a maximal independent set and will not be generated actually. After generating node n_5, $\mathcal{D} \setminus \mathcal{I} = \{t_6, t_8, t_9, t_{10}\}$, so $\texttt{UB}(n_5) = 1.05$ (t_6, t_8 are modified to t_4 and t_9, t_{10} are modified to t_1). $\texttt{UB}(\mathcal{T})$ will also be updated to 1.05.

Algorithm 1: Expansion-based Algorithm for Single FD

Input: A database $\mathcal{D}$, an FD $\varphi : X \rightarrow Y$ and a tree-index
Output: The FT-consistent database $\mathcal{D}'$ with the minimum cost

```
1  T ← {t1};
2  UB(T) = +∞;
3  for level i ← 1 to |D| do
4      for each node I in level i do
5          UB(T) ← min(UB(T), UB(I));
6      for each node I in level i do
7          if LB(I) ≤ UB(T) then
8              if t_{i+1} is FT-consistent with I then
9                  Add I ∪ {t_{i+1}} as a child of I;
10             else
11                 Add I as the left child of I;
12                 if FTC(t_{i+1}, I) ∪ {t_{i+1}} is maximal and not
                   reduplicate then
13                     Add FTC(t_{i+1}, I) ∪ {t_{i+1}} as the right
                       child of I;
14 I^B ← the maximal independent set with cost UB(T);
15 for each tuple t1 ∈ D \ I^B do
16     modify t1^φ to its closest value t2^φ ∈ I^B;
17 return D';
```

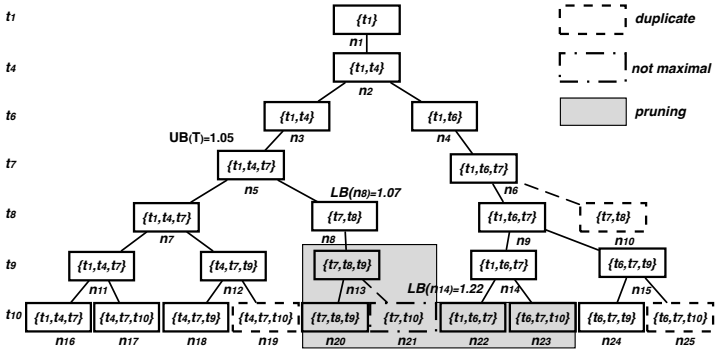


Fig. 3. Expansion-based algorithm

When generating the children of n_8, we already know that $\mathcal{D}_i \setminus \mathcal{I} = \{t_1, t_4, t_6\}$ must be repaired. The minimum repair cost is 1.07 (t_1, t_2, t_3 are modified to t_{10} and t_4, t_5, t_6 are modified to t_8). It is greater than the upper bound $\mathtt{UB}(\mathcal{T})$, so this branch can be pruned. Finally, $\mathcal{I}^B = \{t_1, t_4, t_7\}$. Tuples t_6, t_8 are modified to t_4, and t_9, t_{10} are modified to t_1. □

Tuple grouping. Some tuples may have the same value on attributes in φ, *e.g.*, $t_1^\varphi = t_2^\varphi = t_3^\varphi$. As these tuples must have the same neighbors, we group them together and treat them as a single vertex. However, the edge weight associated with the vertex will be counted multiple times. For example, m tuples are grouped as a vertex t and n tuples are grouped as a vertex t', then the cost of repairing t to t' should be $m \cdot \mathsf{dist}(t, t')$ and the cost of repairing t' to t should be $n \cdot \mathsf{dist}(t, t')$. To this end, we can construct a directed graph $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$, where $\mathcal{V}'$ is the set of tuples with the same value on φ and $\mathcal{E}'$ is the set of directed edges. There is an edge from t to t' if they are not FT-consistent and the weight is $m \cdot \mathsf{dist}(t, t')$ where m is the number of tuples with the same t^φ. We can prove that using $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ to repair $\mathcal{D}$ is equivalent to using $\mathcal{G}(\mathcal{V}, \mathcal{E})$ to repair $\mathcal{D}$. Since $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ has a smaller number of vertices, using $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ to repair $\mathcal{D}$ is more efficient.

Complexity. Given a graph model $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}|$ vertices, $|\mathcal{E}|$ edges, and μ maximal independent sets of the graph, it has been proven in [33] that the algorithm requires processing time and memory space bounded by $O(\mu|\mathcal{V}||\mathcal{E}|)$ and $O(|\mathcal{V}| + |\mathcal{E}|)$ respectively. (This also proves that using $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ to repair $\mathcal{D}$ can improve the complexity.)

3.2 Heuristic: Greedy Algorithm

Due to the high computational complexity of the expansion-based algorithm, we propose a heuristic algorithm that generates an expected best independent set, $\hat{\mathcal{I}}$, to approximate the best independent set $\mathcal{I}^B$. Initially $\hat{\mathcal{I}} = \emptyset$ and we greedily add a tuple with the smallest repairing cost into $\hat{\mathcal{I}}$, until $\hat{\mathcal{I}}$ becomes a maximal independent set.

Initial cost. It first sorts the tuples in order. Given a tuple t_i, if it is added into the independent set, its neighbors will be repaired to t_i, and the initial cost of each tuple t_i is

$$\mathcal{S}(t_i|\emptyset) = \sum_{v \in \mathcal{N}(t_i)} \omega(v, t_i). \tag{7}$$

Initially, we add the tuple with the smallest cost into $\hat{\mathcal{I}}$.

Incremental cost. Given the current independent set $\hat{\mathcal{I}}$ and a tuple $t_i \notin \hat{\mathcal{I}}$, we need to compute the cost of adding t_i into $\hat{\mathcal{I}}$ by considering the following types of tuples.

(1) Consider t_i. As t_i is FT-consistent with $\hat{\mathcal{I}}$, the repairing cost on t_i, whatever t_i is, being added into $\hat{\mathcal{I}}$ is 0.

(2) Consider t_i's neighbor $v \in \mathcal{N}(t_i)$. We consider two cases. (2.1) v is a neighbor of a vertex in $\hat{\mathcal{I}}$, *i.e.*, $v \in \mathcal{N}(t_i) \cap \mathcal{N}(\hat{\mathcal{I}})$, where $\mathcal{N}(\hat{\mathcal{I}}) = \cup_{u \in \hat{\mathcal{I}}} \mathcal{N}(u)$. Now, if t_i is added into $\hat{\mathcal{I}}$, the repair cost of v may be updated, because the repair cost of v given $\hat{\mathcal{I}}$ is $\min_{u \in \hat{\mathcal{I}}} \omega(v, u)$, and the new repair cost is $\min_{u \in \hat{\mathcal{I}} \cup \{t_i\}} \omega(v, u)$, and thus the increased cost is $\min_{u \in \hat{\mathcal{I}} \cup \{t_i\}} \omega(v, u) - \min_{u \in \hat{\mathcal{I}}} \omega(v, u)$. (2.2) v is not a neighbor of a vertex in $\hat{\mathcal{I}}$, *i.e.*, $v \in \mathcal{N}(t_i) \setminus \mathcal{N}(\hat{\mathcal{I}})$. In this case, if t_i is added into $\hat{\mathcal{I}}$, the repair cost of v given $\hat{\mathcal{I}}$ is 0, and the new repair cost is $\omega(v, t_i)$, and the increased cost is $\omega(v, t_i)$.

(3) Consider $v \neq t_i$ is not a neighbor of v. The repair cost is still 0 which is not changed.

Therefore, we can compute the incremental cost as.

$$\mathcal{S}(t_i|\hat{\mathcal{I}}) = \sum_{v \in \mathcal{N}(t_i) \cap \mathcal{N}(\hat{\mathcal{I}})} \min_{u \in \hat{\mathcal{I}} \cup \{t_i\}} \omega(v, u) - \min_{u \in \hat{\mathcal{I}}} \omega(v, u) + \sum_{v \in \mathcal{N}(t_i) \setminus \mathcal{N}(\hat{\mathcal{I}})} \omega(v, t_i). \tag{8}$$

We add the tuple with the largest cost into the independent set $\hat{\mathcal{I}}$. Iteratively, we find the expected best set $\hat{\mathcal{I}}$.

Algorithm. The greedy algorithm is given in Algorithm 2. We first calculate the initial cost of each tuple in $\mathcal{D}$ based on Equation 7 (lines 1-2), and $\hat{\mathcal{I}}$ is an empty set $\emptyset$ by default. We then initialize independent set $\hat{\mathcal{I}}$ by adding the tuple with the smallest initial cost (line 3). Given an independent set $\hat{\mathcal{I}}$, we calculate the incremental cost of each tuple in $\mathcal{D} \setminus \hat{\mathcal{I}}$ that is FT-consistent with $\hat{\mathcal{I}}$ based on Equation 8 and add the tuple t with the smallest incremental cost into $\hat{\mathcal{I}}$ (lines 4-7). Finally we utilize $\hat{\mathcal{I}}$ to repair $\mathcal{D}$ (lines 8-9).

Example 9: Consider $\mathcal{D}$ and φ_1 in Table 1. We illustrate this algorithm with tuples $\{t_1, t_4, t_6, t_7, t_8, t_9, t_{10}\}$. At first, t_7 has the smallest initial cost $\mathcal{S}(t_7|\emptyset) = 0$. So tuple t_7 is first

Algorithm 2: Greedy Algorithm for Single FD

Input: a database $\mathcal{D}$, an FD φ
Output: a FT-consistent database $\mathcal{D}'$

```
for each t ∈ D do
    Compute S(t|∅);
Î ← add the tuple with the smallest initial cost;
while ∃t ∈ D \ Î s.t. t is FT-consistent with Î do
    T ← {t ∈ D \ Î s.t. t is FT-consistent with Î};
    t ← the tuple in T with the smallest incremental cost;
    Î ← Î ∪ {t};
for each tuple t1 ∈ D \ Î do
    modify t1^φ to its closest value t2^φ ∈ Î;
return D';
```

added to $\hat{\mathcal{I}}$ and $T = \{t_1, t_4, t_6, t_8, t_9, t_{10}\}$. Then we calculate the incremental cost of each tuple in T. The neighbor of t_1 is $\mathcal{N}(t_1) = \{t_8, t_9, t_{10}\}$ and $\mathcal{N}(\hat{\mathcal{I}}) \cap \mathcal{N}(t_1) = \emptyset$. Thus, the incremental cost $\mathcal{S}(t_1|\hat{\mathcal{I}}) = 0.56 + 0.11 + 0.6 = 1.27$. Similarly, we compute the incremental cost of other tuples. Tuple t_4 has the smallest incremental cost which is $\mathcal{S}(t_4|\hat{\mathcal{I}}) = 0.2 + 1.14 = 0.34$. So $\hat{\mathcal{I}} = \hat{\mathcal{I}} \cup \{t_4\} = \{t_4, t_7\}$ and $T = \{t_1, t_9, t_{10}\}$. Now, $\mathcal{N}(\hat{\mathcal{I}}) \cap \mathcal{N}(t_1) = \{t_8\}$ and $\mathcal{N}(t_1) \setminus \mathcal{N}(\hat{\mathcal{I}}) = \{t_9, t_{10}\}$. The addition of t_1 cannot decrease the repair cost of t_8, so $\mathcal{S}(t_1|\hat{\mathcal{I}}) = 0.6 + 0.11 = 0.71$ which is smallest. $\hat{\mathcal{I}} = \hat{\mathcal{I}} \cup \{t_1\} = \{t_1, t_4, t_7\}$ and $T = \emptyset$. It terminates. Tuples t_9, t_{10} (resp. t_6, t_8) are modified to t_1 (resp. t_4). □

Complexity. In each iteration, the algorithm chooses a tuple by computing the cost with a complexity of $O(|\mathcal{V}|)$. It terminates in $|\hat{\mathcal{I}}|$ iterations. Thus the complexity is $O(|\hat{\mathcal{I}}||\mathcal{V}|)$. Besides, after tuple grouping, $|\mathcal{V}|$ is much smaller than the number of tuples.

4 Multiple Constraints Solutions

We study the *multiple FDs repairing problem* where multiple FDs are given. We prove that this problem is NP-hard. We also propose an algorithm to find the optimal result and two heuristic algorithms to find approximate results.

4.1 Single FD vs Multiple FDs

We discuss the difference between single FD and multiple FDs in repairing a database.

Repairing target. Given an FD $\varphi : (X \to Y)$, for any tuple t, t^φ could be a target that other tuples can be repaired to. For multiple FDs, we need to revisit the *target* semantics.

Consider two FDs $\varphi_1 : X_1 \to Y_1$ and $\varphi_2 : X_2 \to Y_2$. If φ_1 and φ_2 have no common attributes, for any two tuples t and t', $(t^{\varphi_1}, t^{\varphi_2})$, $(t^{\varphi_1}, t'^{\varphi_2})$, $(t'^{\varphi_1}, t^{\varphi_2})$, $(t'^{\varphi_1}, t'^{\varphi_2})$ are targets. Any tuple can be repaired to one of them. So we independently repair the database on the two FDs, *i.e.*, first repairing $\mathcal{D}$ using one FD and then using another FD to repair. The main reason is that the two FDs have no common attributes and repairing one FD will not affect the other.

Theorem 5: *Given multiple FDs without common attributes, the single FD repairing algorithm can find the optimal repair by independently using the FDs to repair.* □

Proof sketch: If two FDs do not share common attributes, repairing one FD has no influence on the other. So if the repairing cost of each FD is smallest, the total cost of repairing $\mathcal{D}$ is smallest. Thus, the single FD repair algorithm can find the optimal solution by independently using the FDs to repair. □

If φ_1 and φ_2 have a set of overlapping attributes, denoted by $Z = (X_1 \cup Y_1) \cap (X_2 \cup Y_2)$, for any two tuples t and t', if $t^Z = t'^Z$, $t^{\varphi_1} \bowtie_Z t'^{\varphi_2}$ is a target, as any tuple can be repaired to the target; otherwise t^{φ_1} cannot join with t'^{φ_2} and the two tuples cannot generate a target. Moreover, if φ_1 and φ_2 have overlapping attributes, we cannot repair them independently. For example, consider FDs φ_2 and φ_3, and tuple t_5 in Table 1. $t_5^{\varphi_3}$ = (Boston, Main, Manhattan) FT-violates with both (New York, Main, Manhattan) (from t_1, t_2) and (Boston, Main, Financial) (from t_6, t_7). If we only consider φ_3, we repair t_5[District] to Financial with lower cost. Then when considering φ_2, we will modify t_5[State] to MA, and thus repair t_5 to (Boston, Main, Financial, MA). However, if we consider the two FDs jointly, we need to repair t_5 to (New York, Main, Manhattan, NY) whose repairing cost is smallest. Thus, for FDs with common attributes, we need to repair them jointly.

FD graph. We can construct a graph on FDs, where vertices are FDs and if two FDs have common attributes, there is an edge between them. If the graph is not connected, we only consider its connected subgraphs and repair different subgraphs independently. For example, φ_1 and (φ_2, φ_3) are independent. Thus we only need to consider connected FDs.

Valid target. Given n connected FDs, $\varphi_i \in \Sigma : (X_i \to Y_i)$, $t_{j_1}^{\varphi_1} \bowtie t_{j_2}^{\varphi_2} \bowtie \cdots \bowtie t_{j_n}^{\varphi_n}$ are targets, where $1 \le j_i \le |\mathcal{D}|$ and $t_{j_s}^{\varphi_s}$ and $t_{j_t}^{\varphi_t}$ have the same value on their common attributes.

Repairing based on maximal independent sets. For each FD $\varphi_i : X_i \to Y_i$ in Σ, we generate a graph $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$ with vertex set $\mathcal{V}_i$ and edge set $\mathcal{E}_i$. We find every maximal independent set $\mathcal{I}_{\varphi_i}^*$ for $\mathcal{G}_i$. Let $\mathcal{S}_i$ denote the set of maximal independent sets for φ_i and $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 \times \cdots \times \mathcal{S}_{|\Sigma|}$. Given a group of maximal independent sets $\mathcal{I}_\Sigma^* = \{\mathcal{I}_{\varphi_1}^*, \mathcal{I}_{\varphi_2}^*, ..., \mathcal{I}_{\varphi_{|\Sigma|}}^*\}$ in $\mathcal{S}$, we join them to generate the targets. For each tuple x, we use the targets to repair x with the cost of $\mathsf{cost}(x|\mathcal{I}_\Sigma^*)$ which is the sum of repairing cost on each attribute. We can iteratively repair all the tuples in $\mathcal{D}$ using $\mathcal{I}_\Sigma^*$ and get a repaired database $\mathcal{D}'$. The total cost of repairing $\mathcal{D}$ given $\mathcal{I}_\Sigma^*$ is $\mathsf{cost}(\mathcal{D}, \mathcal{D}') = \mathsf{cost}(\mathcal{D}|\mathcal{I}_\Sigma^*) = \sum_{x \in \mathcal{D}} \mathsf{cost}(x|\mathcal{I}_\Sigma^*)$.

Optimal Repairing. We enumerate every $\mathcal{I}_\Sigma^* \in \mathcal{S}$, compute $\mathsf{cost}(\mathcal{D}|\mathcal{I}_\Sigma^*)$, select the group $\mathcal{I}_\Sigma^B$ with the minimal cost, and use the best group $\mathcal{I}_\Sigma^B$ to repair the database.

Theorem 6: *Multiple-FD repair problem is NP-hard.* □

Proof sketch: Consider a special case of the multiple FDs repairing problem: the FDs have no common attributes. In this situation, we can repair $\mathcal{D}$ using the FDs independently. Since the single FD repairing problem is NP-hard, the multiple FDs repairing problem is also NP-hard. □

Next, we will present an expansion-based algorithm for the optimal repair (Section 4.2), and two greedy algorithms for approximate results (Sections 4.3 and 4.4).

4.2 Optimal: Expansion-based Algorithm

We first use the expansion-based algorithm to generate the maximal independent sets on every FD, and then conduct

a Cartesian product on them to generate $\mathcal{S}$. For each group $\mathcal{I}^*_\Sigma \in \mathcal{S}$, we generate the targets and compute the repair cost $\text{cost}(\mathcal{D}|\mathcal{I}^*_\Sigma)$. Next we select the group $\mathcal{I}^B_\Sigma$ with the minimal cost, and use the best group $\mathcal{I}^B_\Sigma$ to repair the database. (We study how to utilize a group $\mathcal{I}^*_\Sigma$ to repair $\mathcal{D}$ in Section 5.) As there are large number of groups, we propose effective pruning techniques to prune unnecessary groups.

Pruning techniques. Similar to the single FD repair algorithm, we build tree-based index for each FD. When we build the index for each constraint, we utilize an effective pruning technique to prune some maximal independent sets that cannot be combined into $\mathcal{I}^B_\Sigma$. We estimate a lower bound of using an independent set $\mathcal{I}_{\varphi_j}$ or its expansion to repair $\mathcal{D}$, denoted by $\texttt{LB}(\mathcal{I}_{\varphi_j})$, and an upper bound $\texttt{UB}(\mathcal{I}_{\varphi_j})$. Given two independent sets $\mathcal{I}_{\varphi_j}$ and $\mathcal{I}'_{\varphi_j}$, if $\texttt{LB}(\mathcal{I}_{\varphi_j}) > \texttt{UB}(\mathcal{I}'_{\varphi_j})$, we prune $\mathcal{I}_{\varphi_j}$ without expanding it.

Lower bound. Given $\mathcal{D}_i = \{t_1, ..., t_i\}$ and an independent set $\mathcal{I}_{\varphi_j}$ of $\mathcal{D}_i$, if we use $\mathcal{I}_{\varphi_j}$ to repair $\mathcal{D}^{\varphi_j}$, the tuples in $\mathcal{D}_i \setminus \mathcal{I}_{\varphi_j}$ must be repaired. We can estimate a lower bound of the repair cost of $\mathcal{I}_{\varphi_j}$:

$$\texttt{LB}(\mathcal{I}_{\varphi_j}|\mathcal{D}^{\varphi_j}) = \sum_{v \in \mathcal{D}_i \setminus \mathcal{I}_{\varphi_j}} \min(\omega(v^{\varphi_j}, u^{\varphi_j}) | u \notin \mathcal{D}_i \setminus \mathcal{I}_{\varphi_j}). \quad (9)$$

For each φ_k, we keep a lower bound $\texttt{LB}_{\varphi_k}$ which is the smallest lower bound of all the independent sets stored in the leaves of the current tree.

A natural idea is to get a lower bound using $\mathcal{I}_{\varphi_j}$ to repair $\mathcal{D}$ by summing up the minimal cost from each φ_j. However, different FDs may have overlapping attributes. To avoid counting them duplicately, we select a set of disjoint FDs, which have no overlap with φ_j, denoted by $F(\varphi_j)$, and get a lower bound,

$$\texttt{LB}(\mathcal{I}_{\varphi_j}) = \texttt{LB}(\mathcal{I}_{\varphi_j}|\mathcal{D}^{\varphi_i}) + \sum_{\varphi_k \in F(\varphi_j)} \texttt{LB}_{\varphi_k}. \quad (10)$$

Upper bound. Given an independent set $\mathcal{I}_{\varphi_j}$, we first find a set of tuples $\mathcal{I}'_{\varphi_j}$ from $\mathcal{I}_{\varphi_j}$ that have no FT-violation for all FDs. Then the upper bound of $\mathcal{I}_{\varphi_j}$ is:

$$\texttt{UB}(\mathcal{I}_{\varphi_j}) = \sum_{v \in \mathcal{D} \setminus \mathcal{I}_\Sigma} \min(\omega(v, u) | u \in \mathcal{I}'_{\varphi_j}). \quad (11)$$

The upper bound UB is the smallest upper bound of all the independent sets. Therefore, an independent set can be safely pruned if $\texttt{LB}(\mathcal{I}_{\varphi_j}) > \texttt{UB} = \min_{\mathcal{I}_{\varphi_j}} \texttt{UB}(\mathcal{I}_{\varphi_j})$.

Tree-based Indexing. We build the tree index $\mathcal{T}_i$ for FD φ_i. We first generate the root with tuple set $\{t_1\}$ for each tree index $\mathcal{T}_i$, compute the lower bound $\texttt{LB}(\{t_1\})$ and the upper bound $\texttt{UB}(\{t_1\})$, and compute the upper bound UB. If $\texttt{LB}(\{t_1\}) \leq \texttt{UB}$, we generate the children of the root. Iteratively for any child of the root with tuple set $\mathcal{I}_{\varphi_i}$, we estimate $\texttt{LB}(\mathcal{I}_{\varphi_i})$ and $\texttt{UB}(\mathcal{I}_{\varphi_i})$ and update UB. If $\texttt{LB}(\mathcal{I}_{\varphi_i}) \leq \texttt{UB}$, we generate the children of $\mathcal{I}_{\varphi_i}$. Iteratively, we can find all the maximal independent sets.

Example 10: Consider $\mathcal{D}$ and $\Sigma = \{\varphi_2, \varphi_3\}$ in Table 1. We can enumerate four maximal independent sets of $\mathcal{D}^{\varphi_2}$: $\mathcal{S}_2 = \{\{t_1^{\varphi_2}, t_6^{\varphi_2}\}, \{t_1^{\varphi_2}, t_8^{\varphi_2}\}, \{t_4^{\varphi_2}, t_5^{\varphi_2}\}, \{t_5^{\varphi_2}, t_8^{\varphi_2}\}\}$ and four maximal independent sets of $\mathcal{D}^{\varphi_3}$: $\mathcal{S}_3 = \{\{t_1^{\varphi_3}, t_3^{\varphi_3}, t_6^{\varphi_3}, t_8^{\varphi_3}\}, \{t_1^{\varphi_3}, t_3^{\varphi_3}, t_6^{\varphi_3}, t_9^{\varphi_3}\}, \{t_3^{\varphi_3}, t_5^{\varphi_3}, t_8^{\varphi_3}\}, \{t_3^{\varphi_3}, t_5^{\varphi_3}, t_9^{\varphi_3}\}\}$. We join $\{t_1^{\varphi_2}, t_6^{\varphi_2}\}$ and $\{t_1^{\varphi_3}, t_3^{\varphi_3}, t_6^{\varphi_3}, t_9^{\varphi_3}\}$, and get four targets $\mathcal{I}_\Sigma = \{$(New York, Main, Manhattan, NY), (New York, Western, Queens, NY), (Boston, Main, Financial, MA), (Boston, Arlingto, Brookside, MA)$\}$. It is the best group to repair the database. Actually, $\{t_4^{\varphi_2}, t_5^{\varphi_2}\}$ and $\{t_5^{\varphi_2}, t_8^{\varphi_2}\}$ are not expanded. Their father node is $\mathcal{I}$ = $\{t_4^{\varphi_2}, t_5^{\varphi_2}\}$ and at that time we know that $t_1^{\varphi_2}, t_6^{\varphi_2}$ must be repaired and UB =3.04. The lower bound LB ($\mathcal{I}$) = 3×dist$(t_1^{\varphi_2}, t_5^{\varphi_2})$ + 3×dist$(t_6^{\varphi_2}, t_8^{\varphi_2})$ = 3.135> UB. Thus it can be safely pruned. □

Algorithm 3: Expansion Algorithm for Multiple FDs

Input: a database D, a set of FDs Σ
Output: a FT-consistent database D'

```
UB ← +∞;
for each constraint φ_i ∈ Σ do
    T_i ← {t_1} and compute LB({t_1}), UB({t_1});
    UB ← min(UB, UB({t_1}));
while exist some independent sets can be expanded do
    I_{φ_i} ← the one with lowest upper bound; expand I_{φ_i};
    for each child of I_{φ_i} do
        compute lower/upper bound, and update UB;
for each constraint φ_i ∈ Σ do
    S_i ← maximal consistent sets for φ_i in T_i;
S ← S_1 × S_2 × ... × S_{|Σ|};
minCost ← +∞;
for each combination I*_Σ = {I*_{φ1}, I*_{φ2}, ..., I*_{φ|Σ|}} in S do
    P ← join the target values of I*_{φ1}, I*_{φ2}, ..., I*_{φ|Σ|};
    cost ← 0;
    for each tuple t contains unresolved values do
        p ← the most simiar one to t^Σ in P;
        cost ← cost + dist_Σ(p, t^Σ);
    if cost < minCost then
        minCost ← cost; bestRepair ← I*_Σ;
repair database D utilizing bestRepair;
return D';
```

Theorem 7: *The expansion-based algorithm finds the optimal repair.* □

Proof sketch: We need first prove that given the maximal independent set $\mathcal{I}^*_\varphi$ of each constraint φ, the join results $\mathcal{P}$ construct the solution space to repair each tuple. Obviously, it is always true. Let us consider a certain database D' which is FT-consistent, and for each tuple $t' \in D'$ and each FD $\varphi \in \Sigma$, $t'^\varphi \in \mathcal{I}_\varphi$. Then t'^Σ must belong to $\mathcal{I}_\Sigma$. Thus, $\mathcal{P}$ contains all possible solution and we only need to choose the one whose repair cost is the smallest for each tuple.
Then, since we enumerate the maximal independent sets for each constraint (only safely prune some of them that cannot lead to the optimal repair) and construct all possible combinations to join them. The expansion-based algorithm can find the optimal repair. □

Complexity. Enumerating all maximal independent sets of Σ requires $O(|\mathcal{V}|^2|\Sigma|)$. The complexity of joining maximal independent sets is $O(\prod_i |\mathcal{S}_i|) = O(|\mathcal{V}|^{|\Sigma|})$. Thus the complexity of multiple FDs repairing is $O(|\mathcal{V}|^{|\Sigma|+1})$.

4.3 Heuristic: Extension of Single FD Greedy Algorithm

Enumerating all maximal consistent sets for each constraint is time-consuming. Actually, we can find the expected optimal independent set of each FD whose corresponding repair cost is the minimum by utilizing the greedy algorithms for single FD. Then we join the target values and repair D based on the join result. The complexity is $O(|\mathcal{V}|^2|\Sigma|)$.

Algorithm 4: Greedy Algorithm for Multiple FDs

Input: a database D, a set Σ of FDs
Output: a FT-consistent database D'

1 **while** $\exists t^{\varphi} \in \mathcal{D} \setminus \hat{\mathcal{I}}_{\Sigma}$ *s.t.* t^{φ} *is FT-consistent with* $\hat{\mathcal{I}}_{\varphi}$ **do**
2 $C \leftarrow \{t^{\varphi} \in \mathcal{D} \setminus \hat{\mathcal{I}}_{\Sigma}$ s.t. t^{φ} is FT-consistent with $\hat{\mathcal{I}}_{\varphi}\}$;
3 $t^{\varphi} \leftarrow$ the one in C with the smallest tuple cost;
4 $\varphi \leftarrow$ the FD s.t. t^{φ} is FT-consistent with $\hat{\mathcal{I}}_{\varphi}$;
5 $\hat{\mathcal{I}}_{\varphi} \leftarrow \hat{\mathcal{I}}_{\varphi} \cup \{t^{\varphi}\}$;
6 update $t'^{\varphi} \in \mathcal{N}(t^{\varphi})$ to t'^{φ}_b;
7 join $\mathcal{I}^*_{\Sigma}$ to get the targets;
8 **for** *each tuple* $t \in \mathcal{D}$ **do**
9 modify t to its closest target;
10 **return** D';

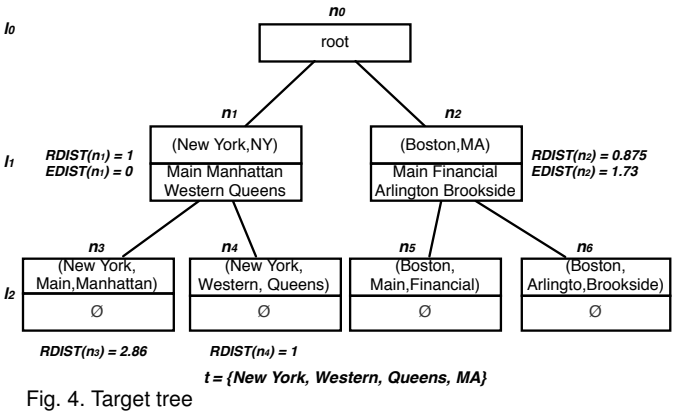


Fig. 4. Target tree

Example 11: Consider $\mathcal{D}$ and $\Sigma = \{\varphi_2, \varphi_3\}$ in Table 1. Utilizing the algorithms for single constraint, $\{t_1^{\varphi_2}, t_6^{\varphi_2}\}$ and $\{t_1^{\varphi_3}, t_3^{\varphi_3}, t_6^{\varphi_3}, t_9^{\varphi_3}\}$ are the corresponding consistent sets of φ_2 and φ_3 in the optimal repair. We join them and get the repair result of Σ without enumerating all combinations. □

4.4 Heuristic: Joint Greedy Algorithm

The single FD greedy algorithm does not consider the relationships between FDs and thus may not generate high quality repairs. To address this problem, we propose a joint greedy algorithm that generates $\hat{\mathcal{I}}_{\varphi_i}$ for each graph $\mathcal{G}_i$.

Tuple cost. If we choose a tuple t and add it to $\hat{\mathcal{I}}_{\varphi_i}$, each tuple t' that is in conflict with t *w.r.t.* φ_i cannot be added to $\hat{\mathcal{I}}_{\varphi_i}$ any more and must be repaired. We should measure the repair cost of t'^{φ_i} as the cost of adding t to $\hat{\mathcal{I}}_{\varphi_i}$. Unlike handling single constraint, we cannot group the tuples that have the same value on attributes in φ_i, because the same value in different tuples might have different repair strategies, *e.g.,* (Boston, NY) in tuple t_5 should be modified to (New York, NY), but repaired to (Boston, MA) in tuple t_{10}. So we need to find the best modification for each tuple.

The tuple t'^{φ_i} can be modified to $t'^{\varphi_i}_c \in \mathcal{N}(t'^{\varphi_i})$. Meanwhile, $t'^{\varphi_i}_c$ cannot be in conflict with any tuple in $\hat{\mathcal{I}}_{\varphi_i} \cup \{t^{\varphi_i}\}$. When we decide how to modify t'^{φ_i}, we are aware of the synchronization across multiple constraints. Assume φ_i is connected to φ_j in FD graph, the impact of modifying t'^{φ_i} on FD φ_j includes two aspects: (1) eliminate some FT-violations for FD φ_j and decrease the repair cost of φ_j, (2) trigger new violations for FD φ_j and increase the repair cost of φ_j. We choose the best one $t'^{\varphi_i}_b$ from $\mathcal{N}(t'^{\varphi_i})$ for tuple t' that eliminates more violations for FD φ_i and FD φ_j and trigger less violations for φ_j. Thus the cost of adding t to $\hat{\mathcal{I}}_{\varphi_i}$ is

$$\mathcal{S}(t^{\varphi_i}|\hat{\mathcal{I}}_{\varphi_i}) = \sum_{v \in \mathcal{N}(t^{\varphi_i})} \mathsf{cost}(v^{\varphi_i}, v_b^{\varphi_i}). \tag{12}$$

Algorithm: The greedy algorithm is shown in Algorithm 4. After computing the repair cost of each tuple, we choose the one whose value is the smallest to add into the independent set (lines 1-6). The tuple that is in conflict with it will be modified to its best choice computed from the above step temporarily. After all tuples in $\mathcal{D}$ are considered, we get a group of maximal independent sets $\mathcal{I}^*_{\Sigma} = \{\mathcal{I}^*_{\varphi_1}, \mathcal{I}^*_{\varphi_2}, ..., \mathcal{I}^*_{\varphi_{|\Sigma|}}\}$. We use them to generate all the targets (line 7) and repair $\mathcal{D}$ (lines 8-9).

Complexity. The time complexity is $O(|\Sigma||\mathcal{V}|^2)$.

Example 12: Consider $\mathcal{D}$, φ_2 and φ_3 in Table 1. There are five different tuple values *w.r.t.* φ_2 and six different tuple values *w.r.t.* φ_3. We show how to compute the tuple cost for $t_1^{\varphi_2}$. If we add $t_1^{\varphi_2}$ (New York, NY) to $\hat{\mathcal{I}}_{\varphi_2}$, tuples t_4, t_5 and t_{10} must be repaired. (New York, MA) in t_4 can be modified to (New York, NY), (Boston, MA) and (Boton, MA). We choose (New York, NY) as it can eliminate three violations *w.r.t.* φ_2 and does not trigger new violations *w.r.t.* φ_3. (Boston, NY) in t_5 and t_{10} can be modified to (New York, NY) and (Boston, MA). (New York, NY) is the best choice for t_5 as it can eliminate four violations *w.r.t.* φ_2 and other four violations *w.r.t.* φ_3. (Boston, NY) in t_{10} should be modified to (Boston, MA) since it does not trigger new violations *w.r.t.* φ_3. □

5 Optimizing Multi-FD Repairing

When joining the maximal independent sets and finding the most similar target for each tuple, it is rather expensive to generate all targets, which may be exponential to the number of tuples. In this section, we describe how to build a tree index to make the above process much faster.

5.1 Target Tree

Target tree. Given a group of maximal independent sets $\mathcal{I}^*_{\Sigma} = \{\mathcal{I}^*_{\varphi_1}, \mathcal{I}^*_{\varphi_2}, ..., \mathcal{I}^*_{\varphi_{|\Sigma|}}\}$, a target tree $\mathcal{T}$ is defined as follows: (1) the root is a dummy node; (2) each node corresponds to an element in each maximal independent set $\mathcal{I}^*_{\varphi_i}$ except the root; (3) the nodes on a path from the root to a leaf correspond to a target (*i.e.,* if the nodes have common attributes, they must have the same values on these attributes); (4) the nodes in the same level come from the same group $\mathcal{I}^*_{\varphi_i}$; (5) each node stores a set of attribute values that appear in the subtree rooted at this node.

Example 13: Consider the database $\mathcal{D}$ in Table 1 and a set of FDs $\Sigma = \{\varphi_2, \varphi_3\}$. The maximal independent sets are $\mathcal{I}^*_{\varphi_2} = \{$(New York, NY), (Boston,MA)$\}$, $\mathcal{I}^*_{\varphi_3} = \{$(New York, Main, Manhattan), (New York, Western, Queens), (Boston, Main, Financial), (Boston, Arlingto, Brookside)$\}$. The target tree is shown in Fig. 4. The elements in $\mathcal{I}^*_{\varphi_2}$ and $\mathcal{I}^*_{\varphi_3}$ are distributed in level l_1 and l_2 respectively. Node n_1 stores (New York, NY) and an attribute-value set {Main, Manhattan, Western, Queens} that contains its descendants' attribute values. Node n_3 is its child node, as node n_3 has the same value "New York" with node n_1 in the common attribute City. The path from the root to node n_3 contains (New York, NY) and (New York, Main, Manhattan). They can be joined into a target (New York, Main, Manhattan, NY). □

Target tree construction. Consider a set Σ of FDs and their corresponding maximal independent sets. Level l_0 is the root which is an empty node. To reduce the size of tree $\mathcal{T}$ and improve the efficiency of searching the tree, the maximal independent sets with fewer elements will be closer to the root, making the root have small fan-outs. To this end, we sort $\mathcal{I}^*_{\varphi_i}$ based on its size in an ascending order, and suppose the sorted sets are $\mathcal{I}^*_{\varphi_1}, \mathcal{I}^*_{\varphi_2}, \cdots, \mathcal{I}^*_{\varphi_{|\Sigma|}}$. Thus the elements in $\mathcal{I}^*_{\varphi_i}$ correspond to nodes in level i. When inserting an element from the independent set into the tree in level l_i ($2 \leq i \leq |\Sigma|$), we first detect whether φ_1 to φ_{i-1} share common attributes with φ_i. We keep a set $\mathcal{A}$ which stores the ancestors of the current node. Since the root is not related to any FD, all elements in $\mathcal{I}^*_1$ are put into $\mathcal{A}$. If φ_1 does not share common attribute with φ_i, the node in $\mathcal{A}$ is replaced by all of its children; otherwise, only the children of the nodes which have same values in the shared attributes are put into $\mathcal{A}$. After inserting all independent sets, we keep the paths from the root to leaf with $|\Sigma|$+1 nodes, because if a path has less than $|\Sigma|$+1 nodes, this path is not a target.

Complexity. Although the worse-case space complexity is $O(\prod_i \mathcal{I}^*_{\varphi_i})$, the actual space is much smaller as many targets share common prefixes.

5.2 Search Target Tree

We discuss how to utilize the target tree to find the best target that has the smallest repairing cost to a given tuple.

We use the best-first search to traverse the target tree to find the best target. After visiting node u, we pick the most promising child v of u with the smallest expected repairing cost $f(v)$ (which will be discussed later) and recursively visit v's children. When we reach a leaf node, we get the real repairing cost between the tuple and the corresponding target. We keep the minimum cost among all the visited targets to the tuple in the process, denoted by $\mathcal{C}_{min}$, and the value $\mathcal{C}_{min}$ can be utilized to prune other nodes (and the subtrees rooted at them). When we visit node u and if $f(u)$ is greater than the minimum cost $\mathcal{C}_{min}$, there is no need to traverse the subtree rooted at u, because $f(u)$ is the smallest cost of modifying tuple t to the targets that are related to the paths from the root to leaf passing node u. After finishing traversing the whole target tree, we can find the best target that has the smallest cost to repair the tuple.

To facilitate the process, we utilize a priority queue to keep the nodes. We first put the root into the queue. Then we pop the node v with the smallest value $f(v)$. We traverse each of v's children, u, compute $f(u)$, and if $f(u) < \mathcal{C}_{min}$, we push u into the queue (prune it otherwise). When we reach a leaf node, we compute the real repairing cost, and if the cost is smaller than $\mathcal{C}_{min}$, we use it to update $\mathcal{C}_{min}$. When the queue is empty, the algorithm terminates.

Computing $f(v)$. For each node v, we compute the lower bound of the repairing costs under v. We consider two types of attribute values.

(1) The attribute values in ancestors of v. As the attribute values are known, we can compute the real repairing cost of the attributes on the nodes from the root to v. Let $\text{RDIST}(v)$ denote this real cost.

(2) The attribute values under v. As there may be more than one leaf node (target) under v and we do not know the exact attribute values, we need to estimate a lower bound. Since we store an attribute-value set in node v that contains all the attribute values in its descendants, we can utilize them to estimate the value. For each attribute in the tuple (not including the attributes corresponding to the ancestors of the node since they have been computed), we find its most similar attribute values in the set and utilize them to estimate the lower bound. Then we sum up the lower bound of every attribute and get an estimated bound $\text{EDIST}(v)$.

Then we can estimate the lower bound of node v:

$$f(v) = \text{RDIST}(v) + \text{EDIST}(v). \tag{13}$$

Algorithm 5: Search Target Tree

```
Input: A tuple t, a target tree T
Output: The most similar target p for tuple t
C_min ← +∞;
minNode ← ∅;
Q ← ⟨T.root, 0⟩;
while Q is not empty do
    u ← Q.dequeue();
    if f(u) < C_min then
        if u is a leaf node then
            C_min ← f(u);
            minNode ← u;
        else
            for each child v of u do
                if f(v)<C_min then Q.enqueue(⟨v, f(v)⟩) ;
p ← the target in the path from root to minNode;
return p;
```

Example 14: Consider the target tree in Fig. 4 and tuple t_4: (New York, Western, Queens, MA). The priority queue is initialized as Q = $\{\langle n_0, 0\rangle\}$ and $\mathcal{C}_{min} = +\infty$. The root n_0 has two child nodes n_1 and n_2. Node n_1 is (New York, NY) and $t_4^{\varphi_2}$ = (New York, MA), so $\text{RDIST}(n_1)$ = dist ('NY','MA') = 1. We can find the value of t_4[Street] and t_4[Distric] in the attribute set stored in n_1, so $\text{EDIST}(n_1)$ = 0. $f(n_1)$ = $\text{RDIST}(n_1)$ + $\text{EDIST}(n_1)$ = 1. In the same way, $\text{RDIST}(n_2)$ = 0.875. The attribute set stored in n_2 is {Main, Financial, Arlington, Brookside}, so $\text{EDIST}(n_2)$ = dist ('Western','Main') + dist ('Queens','Brookside') = 1.73. $f(n_2)$ = $\text{RDIST}(n_2)$ + $\text{EDIST}(n_2)$ = 2.605. Thus Q = $\{\langle n_1, 1\rangle, \langle n_2, 2.605\rangle\}$. We pop n_1 and calculate $f(n_3)$ and $f(n_4)$. Q = $\{\langle n_4, 1\rangle, \langle n_2, 2.605\rangle, \langle n_3, 2.86\rangle\}$. Then we pop n_4. Because n_4 is a leaf node and $f(n_4) = 1 < \mathcal{C}_{min}$. We update $\mathcal{C}_{min}$ to 1. Node n_2 and n_3 are popped in turn. $f(n_3) > f(n_2) > \mathcal{C}_{min}$ so they should be pruned. Finally, t_4 is modified to (New York, Western, Queens, NY). □

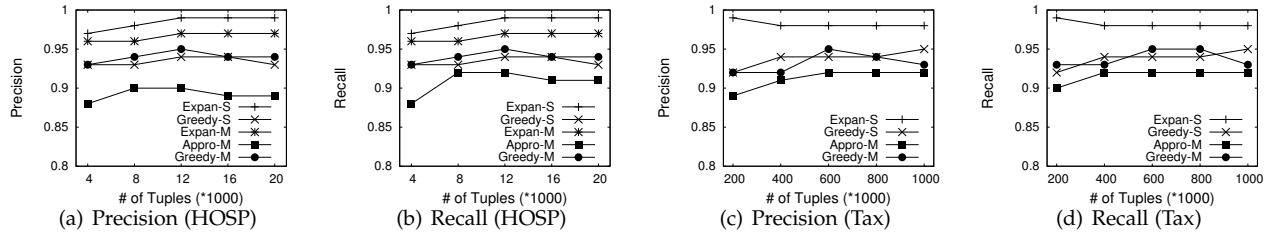
6 Experimental Study

We conducted experiments using real-life and synthetic data for both effectiveness and efficiency study.

6.1 Experimental Settings

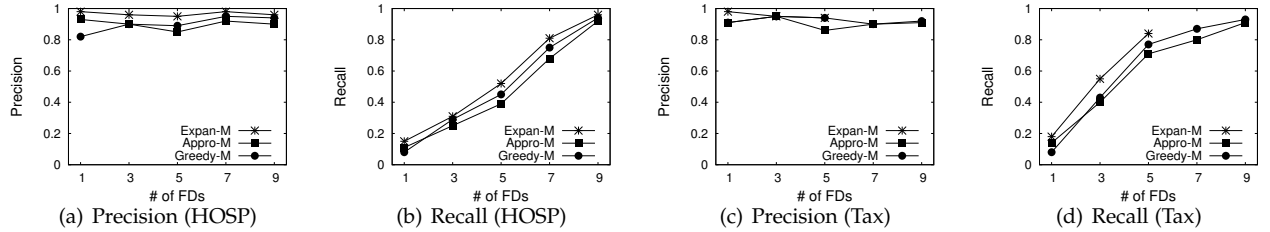
Datasets. (1) *HOSP* was taken from the US Department of Health Services (http://www.hospitalcompare.hhs.gov/). We used 20k records with 19 attributes and 9 FDs. (2) *Tax* was generated by a generator [1]. Each record represented an individual's address and tax information. It had 9 FDs.

1. http://www.cs.utexas.edu/users/ml/riddle/data.html



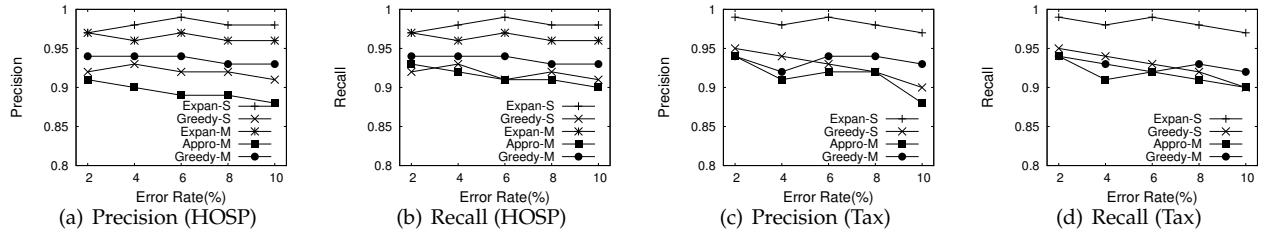
(a) Precision (HOSP) (b) Recall (HOSP) (c) Precision (Tax) (d) Recall (Tax)

Fig. 5. Effectiveness (varying #-tuples N)



(a) Precision (HOSP) (b) Recall (HOSP) (c) Precision (Tax) (d) Recall (Tax)

Fig. 6. Effectiveness (varying #-FDs $|\Sigma|$)



(a) Precision (HOSP) (b) Recall (HOSP) (c) Precision (Tax) (d) Recall (Tax)

Fig. 7. Effectiveness (varying error rates $e\%$)

Noise. Errors were produced by adding noises with a certain rate $e\%$, *i.e.,* the percentage of dirty cells over all data cells *w.r.t.* all FDs. Errors appeared in both left-hand side (LHS) and right-hand side (RHS) of FDs by replacing the attribute values with the values in other tuples. In particular, we randomly added some typos. The rates of RHS errors, LHS errors and typos were equal, *i.e.,* 33.3%.

We used normalized edit distance for string similarity, and normalized Euclidean distance for numeric values and set different distance thresholds τ for different constraints.

Algorithms. We implemented two algorithms for single FD as discussed in Section 3: the expansion algorithm Expan-S and the greedy algorithm Greedy-S; and three algorithms for multiple FDs described in Section 4: the expansion algorithm Expan-M, the extension of single FD greedy algorithm Appro-M, and the greedy algorithm Greedy-M. The algorithm descriptions are shown in Table 2.

Measuring quality. We used precision and recall to evaluate the repairing quality: precision is the ratio of correctly repaired attribute values to the number of all the repaired attributes; and recall is the ratio of correctly repaired attribute values to the number of all erroneous values.

Factors. We evaluated three important factors: #-tuples N, #-FDs $|\Sigma|$, and error rate $e\%$.

#-Tuples: For HOSP, we varied N from 4k to 20k and fixed $e\% = 4\%$. For Tax, we varied N from 200k to 1000k and fixed $e\% = 4\%$. We utilized all FDs for multiple constraints repair.

#-FDs: We varied #-FDs from 1 to all FDs. We fixed $e\% = 4\%$, $N = 8k$ for HOSP and $e\% = 4\%$, $N = 400k$ for Tax.

Error Rate: We varied $e\%$ from 2% to 10%. For HOSP, we fixed $N = 8k$ and for Tax, we fixed $N = 400k$. We utilized all FDs for multiple constraints repair.

Experimental Environment. All methods were written in Java and all tests were conducted on a PC with a 2.60GHz Intel CPU and 64GB RAM, running Ubuntu 12.4. For efficiency, each experiment was run six times, and the average results were reported.

Abbr.	Full Name	Position	Complexity
Expan-S	Expansion-based Algorithm for Single FD	Sec 3.1	$O(\mu\|\mathcal{V}\|\|\mathcal{E}\|)$
Greedy-S	Greedy Algorithm for Single FD	Sec 3.2	$O(\|\mathcal{V}\|\|\hat{\mathcal{I}}\|)$
Expan-M	Expansion Algorithm for Multiple FDs	Sec 4.2	$O(\|\mathcal{V}\|^{\|\Sigma\|+1})$
Appro-M	Extension of Greedy-S for Multiple FDs	Sec 4.3	$O(\|\mathcal{V}\|^2\|\Sigma\|)$
Greedy-M	Greedy Algorithm for Multiple FDs	Sec 4.4	$O(\|\mathcal{V}\|^2\|\Sigma\|)$

TABLE 2
Algorithm Information

6.2 Effectiveness

Figures 5-7 report the effectiveness study by varying #-tuples N, #-FDs $|\Sigma|$, and error rate $e\%$. Note Tax is too large for Expan-M to find the optimal solution, and it could only support less than 5 FDs. In all figures, when utilizing all constraints, the precision and recall of our algorithms were stable within [0.8,0.95]. It tells us that we were able to capture the majority of the true errors in the data within a small number of repairs. The results show that the expansion-based approaches Expan-S and Expan-M had better precision and recall values than the approximate solutions Greedy-S, Appro-M and Greedy-M as expected. Greedy-M outperformed Appro-M because it considered the synchronization across multiple constraints. Since Greedy-S and Greedy-M had limited quality loss, we could utilize them to repair the databases approximately.

#-Tuples. Figure 5 shows that as N increased, the precision and recall of all algorithms remained stable. This tells us that our algorithms could work well with larger data.

#-FDs. Figure 6 shows that as #-FDs increased, the recall of our methods increased. This is because more constraints can be utilized to detect errors. Appro-M did not consider the impact between constraints but only find the approximate optimal solution in each of them. Instead, Greedy-M measured the change of violations with respect to the connected FDs. That is why the effectiveness of Greedy-M is higher than Appro-M. When handling a single FD, Appro-M outperformed Greedy-M. Like Greedy-S, Appro-M had incremental cost for each tuple when the independent set

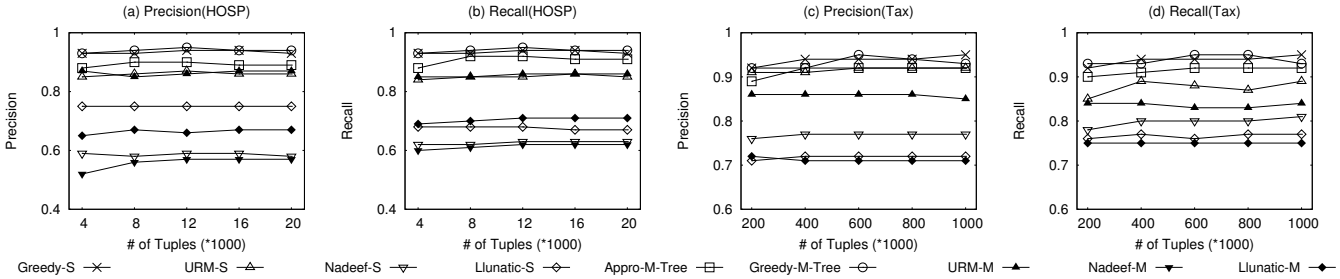


Fig. 11. Quality Comparison (varying #-tuples N)

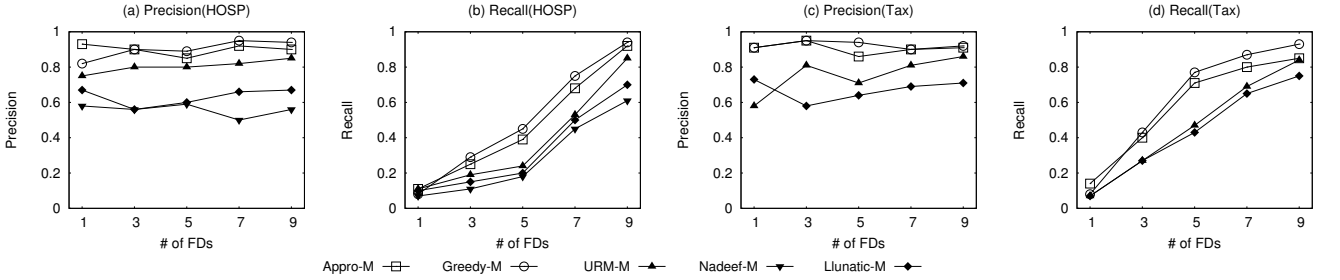


Fig. 12. Quality Comparison (varying #-FDs $|\Sigma|$)

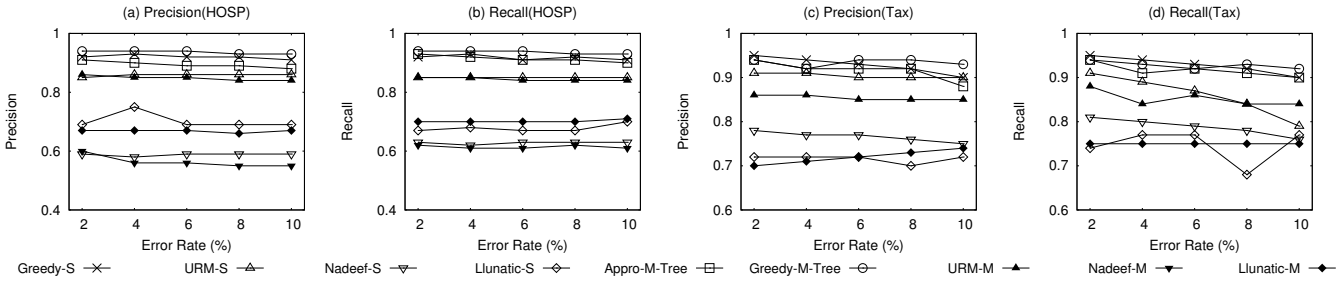
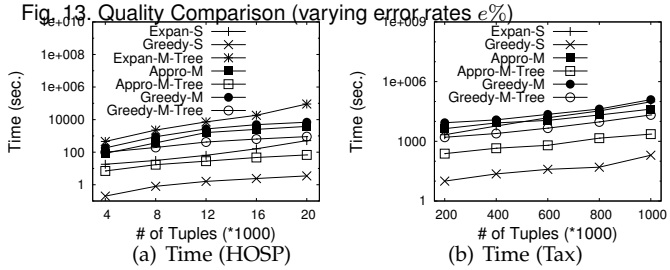
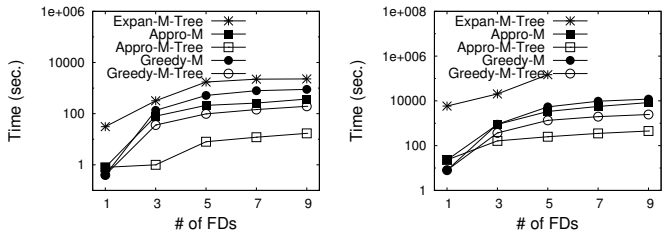


Fig. 13. Quality Comparison (varying error rates $e\%$)



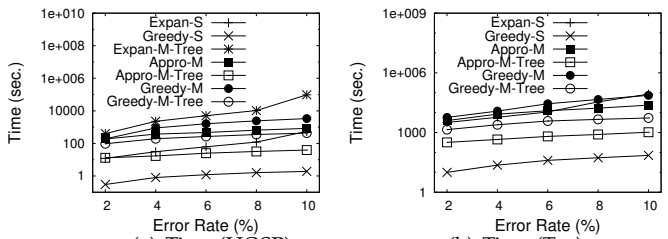
(a) Time (HOSP) (b) Time (Tax)

Fig. 8. Efficiency (varing #-tuples N)



(a) Time (HOSP) (b) Time (Tax)

Fig. 9. Efficiency (varying #-FDs)



(a) Time (HOSP) (b) Time (Tax)

Fig. 10. Efficiency (varying error rates $e\%$)

was not empty. On the contrary, without connected FDs, Greedy-M only had initial cost for each tuple. Expan-M had the best precision and recall, as expected. Note that as #-FDs increased, the quality sacrifice was not signifiant as our methods found high-quality repairs.

Error Rate. Figure 7 shows that when $e\%$ increased, the precision and recall moderately decreased, because a larger error rate made more noise and the problem became harder.

6.3 Efficiency

We evaluated the efficiency of Expan-S, Greedy-S, Expan-M, Greedy-M, Appro-M. For Expan-M, Greedy-M, Appro-M, we could utilize the target tree index to improve efficiency (but cannot improve quality), and thus we compared them with those with the target tree (Expan-M-`Tree`, Greedy-M-`Tree`, Appro-M-`Tree`). As Expan-M was too slow, we only showed the results with tree index Expan-M-`Tree`.

Impact of Target Tree Pruning. With the join target tree, the efficiency was significantly improved. As shown in Fig. 8, Appro-M-`Tree` outperformed Appro-M, and Greedy-M-`Tree` was much better than Greedy-M. As shown in Fig. 9, when varying #-FDs, if there was a single FD, the runtime of Appro-M and Appro-M-Tree were the same. As #-FDs increased, however, the pruning technique can significantly improve the efficiency. For HOSP, when $N = 8k$, $e\% = 4\%$ and #-FDs = 9, the runtime of Appro-M was 372 seconds but 17 seconds for Appro-M-Tree. For Tax, when $N = 400k, e\% = 4\%$ and #-FDs =9, the runtime of Appro-M was 8380 seconds but only 450 seconds for Appro-M-Tree. The results show the superiority of our pruning techniques.

#-Tuples. Figure 8 shows the runtime as #-tuples increases. Comparing with all other algorithms, it shows that the runtime of Greedy-M grew aggressively. It is because the greedy algorithm for multiple FDs needed to compute the

	HOSP			Tax		
	Precision	Recall	Time(s)	Precision	Recall	Time(s)
Greedy-S	0.93	0.93	0.8	0.94	0.94	23
URM-S	0.86	0.85	0.3	0.91	0.89	12
Nadeef-S	0.58	0.62	2	0.77	0.80	14388
Llunatic-S	0.75	0.68	2	0.72	0.77	438
Appro-M-Tree	0.90	0.92	17	0.92	0.91	450
Greedy-M-Tree	0.94	0.94	194	0.92	0.93	2467
URM-M	0.85	0.85	20	0.86	0.84	3520
Nadeef-M	0.56	0.61	358	N.A.	N.A.	N.A.
Llunatic-M	0.67	0.70	24	0.71	0.75	3750

TABLE 3
Algorithm Comparison

best repair for every tuple. The running time of Appro-M also increased evidently, since there were more target values to join and the algorithm computed the nearest pattern from the join result for each tuple.

#-FDs. Figure 9 shows the runtime by varying #-FDs. As $|\Sigma|$ increased, the synchronization across multiple constraints was dominating the time so the calculation in Greedy-M became slower. The runtime of Appro-M increased as the targets of more constraints needed to be joined. We also added two more FDs on HOSP. Appro-M and Greedy-M took 455s and 1088s respectively. The efficiency was linear with the number of FDs and increasing FDs lowered down efficiency, which is consistent with the complexity in Section 5.

Error Rate. Figure 10 shows the runtime by varying $e\%$ from 2% to 10%. When $e\%$ increased, the runtime of Greedy-M for large datasets increased evidently. This is because more patterns t^{φ} were generated with high error rate and needed to be considered whether to be added into the independent set. Moreover, more tuples were injected as errors and needed to find the best way of repair. The runtime of Appro-M grew slowly. This is because even though $e\%$ increased, the target values of each constraint did not change, and the join time did not increase. The runtime of Appro-M was more stable when varying $e\%$, as expected.

6.4 Comparison with Existing Methods

We compared with NADEEF [13], Unified Repair Model (URM) [8] and Llunatic [20] for FD repairs. For URM, to ensure a fair comparison, we implemented it with only data repair option without constraint repair. For Llunatic, we chose the frequency cost-manager and *Metric 0.5* was used to measure the repair quality (for each cell repaired to a variable, it was counted as a partially correct change).

Effectiveness. Figures 11-13 show the quality result. For better presentation, the result of comparisons is also shown in Table 3 ($N = 8k$, $|\Sigma| = 9$ and $e\% = 4\%$ for HOSP and $N = 400k$, $|\Sigma| = 9$ and $e\% = 4\%$ for Tax). Note NADEEF-M could not support the larger dataset Tax. Our algorithms outperformed other methods. We injected the following errors: LHS/RHS active domain errors and typos. NADEEF was the algorithm that only repairs RHS errors. It could modify LHS values only when these attributes also appear on RHS of other FDs. URM was the algorithm that handled all types of errors. It defined core pattern (whose frequency is larger than the threshold), deviant pattern (whose frequency is smaller than the threshold) and always modifies deviant pattern to core pattern to minimize the description length. However, (1) frequency threshold is not enough to measure whether a pattern is correct value. That is why the Greedy-S outperforms URM. (2) URM orders FDs

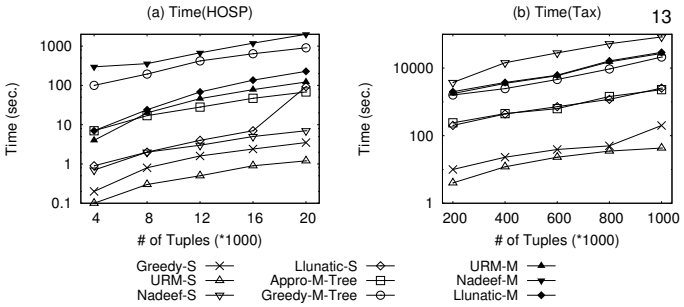


Fig. 14. Efficiency Comparison (varying #-tuples N)

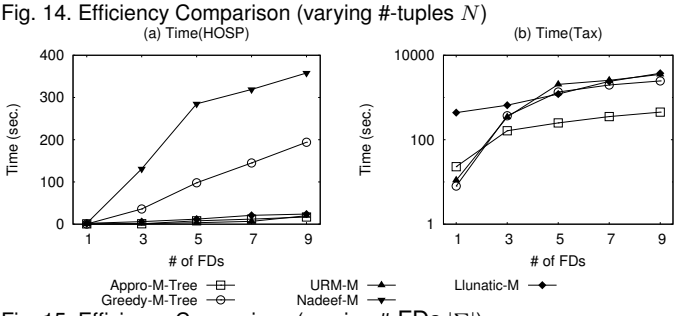


Fig. 15. Efficiency Comparison (varying #-FDs $|\Sigma|$)

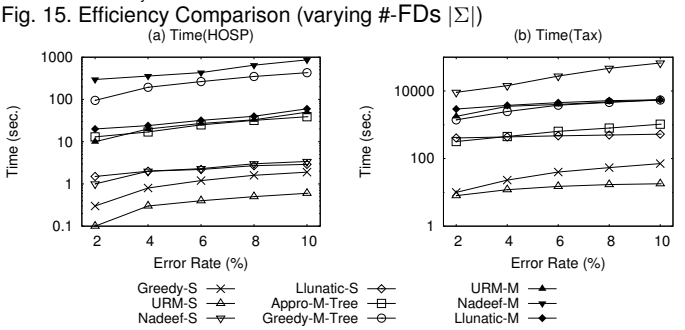


Fig. 16. Efficiency Comparison (varying error rates $e\%$)

and handles them one by one. But obviously, to get the optimal repair of each tuple, different tuples should have different repair orders of FDs. (3) in URM, the same deviant pattern in different tuples are modified to the same core pattern. And if this modification of deviant pattern cannot decrease the description length, the algorithm cannot touch them. However, the same pattern in different tuples should have different ways of repairs as discussed in Section 4.4. Naturally, the precision and recall of URM were lower than our algorithms. Llunatic handled all types of errors. However, it modified a cell to variable to indicate such value was currently unknown and might be resolved into a constant by asking users. The cell can only be partially repaired.

Efficiency. Figures 14-16 show that Greedy-S ran faster than NADEEF and Llunatic but slower than URM, as URM only computed core patterns based on the frequency but Greedy-S calculated the cost by adding a tuple into the independent set. NADEEF and Llunatic were slow as they were DBMS-based algorithms. NADEEF was a generalized data cleaning system as a tradeoff of sacrificing some efficiency [13] for generalization. When handling multiple FDs, Appro-M with pruning ran fastest. After getting the repair targets in each constraint, Appro-M joined them and computed the best repair for each tuple utilizing an effective tree index. Even though URM can get the targets much faster, it needed a lot of calculations to decide how to repair every deviant tuple and maintain the changes of core and deviant patterns. On the larger dataset Tax, Greedy-M ran faster than other algorithms except in high error rate. This is because Greedy-M chose a pattern to add into the independent set from all patterns of all constraints and computed the best repair

for each related tuple. The increase of error rate made the decision harder especially in Greedy-M.

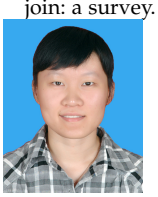
7 Conclusion

We have proposed a revised automatic data repairing problem, using distance-based metrics for error detection and data repairing. We have devised a fault-tolerant data repairing framework. We have identified the complexity of the revised problem, and presented effective exact and approximate data repairing algorithms to compute repairs. Our experimental results with real-life and synthetic data have verified effectiveness and efficiency of our algorithms.

Acknowledgement. Guoliang Li was supported by 973 Program of China (2015CB358700), NSF of China (61373024, 61632016, 61422205, 61521002), Shenzhou, Tencent, Tsinghua TNList, FDCT/116/2013/A3, MYRG105 (Y1-L3)-FST13-GZ.

References

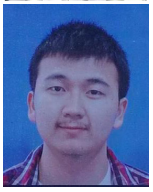
[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 3(4-5), 2003.
[2] L. Berti-Equille, T. Dasu, and D. Srivastava. Discovery of complex glitch patterns: A novel approach to quantitative data cleaning. In *ICDE*, pages 733–744, 2011.
[3] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*, 2011.
[4] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1), 2010.
[5] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David. Modeling and querying possible repairs in duplicate detection. In *VLDB*, 2009.
[6] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.
[7] L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *VLDB*, 2007.
[8] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, 2011.
[9] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2), 2005.
[10] X. Chu, M. Ouzzani, J. Morcos, I. F. Ilyas, P. Papotti, N. Tang, and Y. Ye. KATARA: reliable data cleaning with knowledge bases and crowdsourcing. *PVLDB*, 8(12), 2015.
[11] X. Chu, P. Papotti, and I. Ilyas. Holistic data cleaning: Put violations into context. In *ICDE*, 2013.
[12] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.
[13] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.
[14] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.
[15] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.
[16] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.
[17] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.
[18] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.
[19] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353), 1976.
[20] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The llunatic data-cleaning framework. *PVLDB*, 6(9), 2013.
[21] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.
[22] J. Hellerstein. Quantitative data cleaning for large databases. In *Technical report, UC Berkeley*, 2008.
[23] S. Kolahi and L. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.
[24] N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In *ICDE*, 2009.
[25] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *IEEE Trans. Knowl. Data Eng.*, 28(9):2296–2319, 2016.
[26] R. Mao, H. Xu, W. Wu, J. Li, Y. Li, and M. Lu. Overcoming the challenge of variety: big data abstraction, the next evolution of data management for AAL communication systems. *IEEE Communications Magazine*, 53(1):42–47, 2015.
[27] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.
[28] F. Naumann, A. Bilke, J. Bleiholder, and M. Weis. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.*, 29(2), 2006.
[29] V. Raman and J. M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, 2001.
[30] B. Saha and D. Srivastava. Data quality: The other face of big data. In *ICDE*, 2014.
[31] Z. Shang, Y. Liu, G. Li, and J. Feng. K-join: Knowledge-aware similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(12):3293–3308, 2016.
[32] S. Song and L. Chen. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.*, 36(3):16, 2011.
[33] A. H. e. a. Tsukiyama S, Ide M. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 1977.
[34] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.
[35] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, pages 457–468, 2014.
[36] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*, pages 553–564, 2013.
[37] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5), 2011.
[38] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.



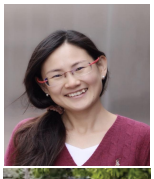
Shuang Hao received the bachelor's degree from the Institute of Software Engineering, Shandong University, China. She is currently working toward the PhD degree in the Department of Computer Science, Tsinghua University, Beijing, China. Her research interests include data cleaning and data integration.

Nan Tang is a senior scientist at Qatar Computing Research Institute, HBKU, Qatar Foundation, Qatar. He received the PhD degree from the Chinese University of Hong Kong in 2007. He has worked as a research staff member at CWI, the Netherlands, from 2008 to 2010. He was a research fellow at University of Edinburgh, from 2010 to 2012. His current research interests include data curation and data streams.

Guoliang Li is currently working as an associate professor in the Department of Computer Science, Tsinghua University, Beijing, China. He received his PhD degree in Computer Science from Tsinghua University, Beijing, China in 2009. His research interests mainly include data cleaning and integration, spatial databases and crowdsourcing.



Jian He received the bachelor's degree from the Department of Computer Science, Huazhong University of Science and Technology, China. He is currently a post graduate student at the Department of Computer Science in Tsinghua University. His research interests mainly include data cleaning and data integration.



Na Ta received her M.S. degree in Computer Science from Tsinghua University in 2007. She is currently a PhD student at the Department of Computer Science in Tsinghua University. Her main research interests include urban computing and spatial databases.

Jianhua Feng received his B.S., M.S. and PhD degrees in Computer Science from Tsinghua University. He is currently working as a professor of Department Computer Science in Tsinghua University. His main research interests include large-scale data management and analysis and spatial database.